

DESIGN, ANALYSIS, AND IMPLEMENTATION
OF PARALLEL EXTERNAL SORTING ALGORITHMS

by

Dina Bitton Friedland

Computer Sciences Technical Report #464

January 1982

DESIGN, ANALYSIS, AND IMPLEMENTATION
OF PARALLEL EXTERNAL SORTING ALGORITHMS

by

DINA BITTON FRIEDLAND

A thesis submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1981

ACKNOWLEDGMENTS

Being a graduate student at the University of Wisconsin-Madison was a stimulating experience. I feel that I was fortunate to pursue my studies in this environment. Several faculty members have helped me to get started on this research, by their high quality teaching and their advice. In particular, I would like to thank Robert Cook and Raphael Finkel for all I have learned from them. Jim Goodman and Diane Smith have often provided helpful answers to my questions. My friend and colleague, Haran Boral listened, read numerous drafts and made helpful suggestions. Most of all, I would like to express my gratitude to my advisor, David DeWitt, who has provided me with generous support and encouraged me several times along the way. He provided constructive criticism, taught me to quantify my ideas (by requesting numbers!) and translated this thesis to readable English.

Finally, I would like to thank my husband and my children for their patience. They have endured Madison's winter, while waiting for me to finish, and have suffered from my bad moods every time I was too busy.

ABSTRACT

In recent years, parallel sorting has been an active field of research. However, parallel sorting algorithms developed up to date cannot be used to sort a large file, because they are internal sorting algorithms. Moreover, it is not possible to implement the fastest among these algorithms with current technology.

This thesis investigates the topic of parallel external sorting. Several new algorithms are presented and analyzed, using a comprehensive cost model that includes computation, communication, and I/O. The I/O cost factor is especially critical for external sorting algorithms. While an extensive literature exists that addresses computation and communication issues in parallel processing, the impact of I/O on the performance of parallel algorithms has not received adequate consideration. We propose several criteria that can measure this impact, when the mass-storage device is a parallel read/write disk.

As a major application of parallel external sorting, we consider the execution of complex database operations. In particular, we propose to use a modified merge-sort as a method for eliminating duplicate records in a large file. A combinatorial model is developed to provide an accurate

estimate for the cost of the duplicate elimination operation (both in the serial and the parallel cases).

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGMENTS iv

TABLE OF CONTENTS v

FIGURES vii

1. INTRODUCTION 1

 1.1. Motivation 1

 1.2. Overview of the sorting problem 3

 1.2.1. Internal sorting 4

 1.2.2. External sorting 7

 1.3. Organization of dissertation 8

2. TAXONOMY OF PARALLEL SORTING 11

 2.1. Parallelizing serial sorting algorithms 12

 2.1.1. The odd-even transposition sort 15

 2.1.2. A parallel tree-sort algorithm 16

 2.2. The network sorting algorithms 19

 2.2.1. Sorting networks 19

 2.2.2. Sorting on an SIMD machine 31

 2.3. The shared memory algorithms 34

 2.4. Parallel tape sorting algorithms 41

 2.5. Summary 44

 2.5.1. Parallel architecture constraints 44

 2.5.2. Further aspects of parallel sorting ... 47

3. A MODEL FOR PARALLEL EXTERNAL SORTING 50

 3.1. Problem statement and terminology 52

 3.1.1. Record and key sorting 52

 3.1.2. Physical order on a disk 53

 3.1.3. Parallel internal and external sorting 57

 3.2. A class of multiprocessors 59

 3.2.1. Processor synchronization 60

 3.2.2. The I/O device 62

 3.3. A cost evaluation model 66

 3.3.1. Computation cost 67

 3.3.2. Communication cost 69

 3.3.3. I/O cost 70

4. PARALLEL EXTERNAL SORTING ALGORITHMS 74

 4.1. Building blocks 77

 4.1.1. Selection, enumeration, and bucket sort 78

4.1.2. Iterated merging	79
4.2. The pipelined merge-sort	85
4.2.1. Description of the algorithm	86
4.2.2. Implementation	88
4.2.3. Analysis	90
4.3. The parallel binary merge-sort	94
4.3.1. Description and implementation	94
4.3.2. Analysis	98
4.4. The external block bitonic sort	101
4.2.1. Description of the algorithm	102
4.2.2. Implementation	105
4.2.2. Analysis	107
4.5. The pipelined selection sort	109
4.5.1. Description of the algorithm	110
4.5.2. Implementation	111
4.5.3. Analysis	112
4.6. The broadcast-enumeration sort	116
4.6.1. Description and implementation	117
4.6.2. Analysis	119
4.7. Performance comparison of the five sorting algorithms	123
4.7.1. Description of the parameters values ..	124
4.7.2. Results of the experiments	127
4.7.3. Evaluation of I/O time	134
4.8. Conclusion	136
5. DUPLICATE ELIMINATION	141
5.1. Duplicate elimination in a relational DBMS ..	143
5.2. Algorithms for duplicate elimination	148
5.3. A combinatorial model	155
5.3.1. Combinations of a multiset	156
5.3.2. Average number of distinct elements ...	158
5.4. Cost analysis	163
5.4.1. Serial duplicate elimination	164
5.4.2. Parallel duplicate elimination	167
5.4.3. Non-uniform duplication	169
5.5. Conclusion	170
6. CONCLUSION	172
6.1. Contributions	172
6.2. Future research	174
6.2.1. Parallel sorting in database machines ..	174
6.2.2. Control cost of parallel algorithms ...	177
REFERENCES	179

FIGURES

1	Tree selection sort	17
2	A comparison-exchange module	21
3	The odd-even merge rule	22
4	The bitonic merge rule	24
5	Batcher's bitonic sort for 8 elements	27
6	Stone's modified bitonic sort	29
7	Stone's architecture for the bitonic sort	30
8	Indexing schemes for an array processor	33
9	Muller's sorting network	36
10	Even's tape sorting algorithm	43
11	Processors required, and computation time	45
12	Physical Order on a Disk	55
13	A Multiprocessor Architecture	64
14	An Alternative Multiprocessor Architecture	65
15	External 2-way merge sort	81
16	Buffers contents after 4 comparison-move steps	82
17	Forecasting the next input page	84
18	The pipelined merge-sort algorithm	87
19	Architecture for the pipelined merge sort	91
20	Architecture for the parallel binary merge-sort	95
21	The parallel binary merge-sort algorithm	96
22	The Block Bitonic Sort Algorithm	104
23	Processor's memory for external block bitonic sort	106
24	Architecture for pipelined selection sort	113
25	Processor's memory for pipelined selection sort	114
26	Architecture for broadcast-enumeration sort	120
27	Computation time of the algorithms	128
28	Ideal track transfer time in milliseconds	135
29	Best case execution time	137
30	Worst case execution time	138
31	Modified Merge	152
32	Number of Distinct Records in Successive Runs	162
33	Cost of serial duplicate elimination	166
34	Early termination of modified merge	168
35	Cost of parallel duplicate elimination	169

CHAPTER 1

INTRODUCTION

1.1. Motivation

It is estimated that over 25% of computer time is spent on sorting [Knut73], when all users are taken into account. This estimate might be even too low for a data processing environment, where sorting of large files is often performed. There are two reasons that explain why such a high percentage of computer resources are utilized for sorting. The first is that sorting is often required, either to deliver to a user a well organized output, or as an intermediate step in the execution of a complex algorithm. The second is that sorting is a time consuming operation, even when a very efficient sort method is used. Therefore, there are two possibilities in order to reduce the amount of time devoted to sorting: find alternatives to sorting and develop faster sorting methods.

It may seem that advances in computer technology could eliminate, or at least reduce significantly, the use of sorting as a tool for performing other operations. For example, when sorting is used in order to facilitate searching, one may advocate that the advent of associative memories will make sorting unnecessary. However,

associative stores are much too expensive for widespread usage, especially when large volumes of data are involved. In the case that sorting is required for the sole purpose of ordering data, the only alternative is to develop faster sorting algorithms. Many serial sorting algorithms that perform in optimal time (that is sort n items in time $O(n \log n)$) are known. The introduction of parallel processing has added a new dimension to research on sorting algorithms. With the use of multiple processors, sorting time can be reduced, at least in theory, to $O(\log n)$. During the past decade, numerous results on parallel sorting have been published. In particular, optimal parallel sorting algorithms have been developed [Hirs78, Prep78] for a theoretical parallel processor model.

In this thesis, we investigate the use of parallel sorting as a tool for an efficient implementation of several database operations on a multiprocessor database machine. Our research was first motivated by our work on the database machine DIRECT [DeWi79]. When looking for a way to eliminate duplicate tuples, we realized that none of the known parallel sorting algorithms were of any value. The main reason was that they are all internal sorting algorithms, that is they assume that the source data can reside in the multiprocessor's memory during the sort operation. Therefore, it became clear that database

machines require a "parallel external sort" facility. The first part of this thesis is devoted to parallel external sorting algorithms. The second part explores the applications of sorting in relational database machines. In particular, we propose to use sorting as a tool to perform the duplicate elimination operation, and justify this method by developing a comprehensive cost evaluation model.

1.2. An overview of the sorting problem

Sorting is defined as the process of rearranging a sequence of items into ascending or descending order. A basic sorting operation deals with items which are all key, that is the order is defined on the items themselves. A more general sorting procedure deals with records where one of the record fields or the concatenation of several fields constitute the key according to which the records are to be sorted. In a database environment, sorting mostly refers to record sorting. The implications of dealing with record sorting are significant in terms of storage and data movement, since typically a record contains several hundred bytes, while the key may only be a few bytes long.

When the data set to be sorted can be stored entirely in main memory, the sorting algorithm is called internal. The algorithm proceeds by comparing and moving pairs of records within the memory. On the other hand, when the

size of the data set is larger than main memory, an external sorting algorithm must be utilized. Internal and external sorting algorithms are also termed as array sorting and file sorting algorithms, respectively. This is because internal sorting algorithms start with reading the data set into a contiguous area of main memory (an "array"), while external sorting algorithms are mainly used to sort large data files that are stored on a mass storage device.

There are numerous efficient serial sorting algorithms [Knut73, Lori71]. Each has certain advantages and disadvantages that must be weighed in the light of the amount of resources available for a particular application. The tradeoffs between execution time, complexity of the algorithm and storage requirements are well known. In the following two sections we will briefly describe some of the criteria that are commonly used to evaluate a sorting algorithm. The primary purpose of this description is to set the grounds for our research on parallel sorting. It is by no means intended to be a survey of serial sorting.

1.2.1. Internal sorting

The dominant cost of an internal sorting algorithm is the cost of comparing and moving elements in main memory. Therefore, an efficient algorithm must minimize this cost.

Optimal sorting algorithms require $O(n \log n)$ comparisons and moves. However, in practice, algorithms that theoretically require $O(n^2)$ time (such as the commonly used bubble sort), are often preferred. In addition to the number of comparisons and moves, an internal sorting algorithm is judged by several other characteristics. It is often the case that a sorting scheme can take advantage of a particular distribution of the initial data. For this reason, empirical studies are as important as analytical results since they can supply an indication of average, best and worst case behavior. In addition to closed analytical formulas, any comprehensive survey of sorting reports execution times measured by experiments on real data and fast computers [Knut73, Wirt76 p.125]. Another crucial factor in evaluating an internal sorting algorithm is the amount of main memory that is required to execute it. First, the program itself may require a substantial amount of storage, when the algorithm is complex. Second, a sorting scheme may necessitate a substantial amount of work space in memory, in addition to the source data space. This is because instead of being sorted "in situ", records must be moved to additional storage locations.

Sorting algorithms requiring a larger work space (e.g. a work space as large as the data space itself) are usually simpler and faster than sorting algorithms that require

only a small amount of memory cells in addition to the data space. The tree selection sort and the heap sort [Knut73] are examples of two algorithms that illustrate this idea. To sort $n=2^m$ values, the tree selection algorithm allocates space for a binary tree structure with n leaf nodes. The n values are first read into the leaf nodes locations, then moved up the empty interior nodes as the algorithm proceeds. The highest of two sibling nodes values is moved up to the parent node, until the maximum value reaches the root node. For the heap sort algorithm, on the other hand, a tree with a total of n nodes is used. The values to be sorted are read into all the tree nodes (leaves and interior nodes). Then, the values locations are interchanged as required for a "heap" structure. Again, the maximum value is propagated to the root, but all the values must be located so that a parent node always hold a value higher than its two children nodes. After the maximum value reaches the root, it is removed and replaced by one of the leaf nodes value. Then, again, the $n-1$ values remaining in the tree are permuted so that they constitute a heap. Thus, the heap sort sorts the n values in place, while the tree selection sort requires approximately twice as much memory. The tradeoff between using more memory or being faster is referred as the time-space tradeoff, and an internal sorting scheme is usually judged by its space requirements, as well as by its execution time.

1.2.2. External sorting

Since a large file cannot fit in main memory, an external sorting method must read a section of the file from secondary storage (such as disks or tapes), process it, and write it back to secondary storage before another section can be processed. Because every record must be compared to every other record, sorting cannot be performed in a single pass over the file. Thus, the cost of I/O transfers becomes a significant factor in the design and the evaluation of an external sorting scheme.

Merging of sorted lists is the basic building block for external sorting. The simplest external sort algorithm is a 2-way merge sort: It consists of iteratively merging pairs of lists of length 2^{k-1} (also called "runs") into a sorted list of length 2^k for $k=1, \dots, \log n$. Variations of this scheme include the Multiway Merge (in an N-way merge, N runs instead of 2 are merged together at each step), and the N-way Balanced Merge where the output runs are written alternately on different files [Knut73, Wirt76]. In practice, one does not use a pure merge-sort algorithm. Typically, the file is initially partitioned into equal sections small enough to fit in main memory. Then, these sections are sorted using a fast internal sorting algorithm. Finally, the merge sort is initiated.

The main cost factor for an external sorting algorithm is the I/O time, since efficient accessing to external files is severely limited by the physical characteristics of the mass storage device. In addition to the number of block transfers, an efficient external sorting algorithm must minimize the access time to the intermediate data files it creates.

1.3. Organization of dissertation

In Chapter 2, the literature on parallel sorting is surveyed and a taxonomy of parallel sorting is proposed. Parallel sorting algorithms are classified according to the type of multiprocessor architecture they apply to. We point out that most existing algorithms should be considered as "array sorting" algorithms, and that parallel file sorting algorithms have not been previously investigated.

In Chapter 3, the concept of "parallel external sort" is introduced and a model for parallel external sorting is presented. This model specifies a class of multiprocessors that have architectural features for efficiently supporting database operations. It also includes the specification of several cost parameters that can be used for analyzing the performance of parallel external sorting algorithms. In particular, criteria for measuring the amount of I/O

activity in a parallel computer environment are defined. In Chapter 4, several new parallel external sorting algorithms are presented. For those algorithms that are based on iterated merging, we propose to use as a building block a new serial external merge procedure, that has the advantage of synchronizing the read requests.

Among the external sorting algorithms proposed, two are parallel versions of the serial 2-way external merge and one is an extension of Batcher's bitonic sort [Batc68]. Each algorithm is analyzed in detail, according to the methodology developed in the previous chapter. A performance comparison of the algorithms concludes this chapter.

Chapter 5 is devoted to the topic of duplicate elimination. Several methods for eliminating duplicate records in a large file are described. We contend that a modified external sort is an efficient way to perform duplicate elimination. In order to estimate the cost of this modified sort (and thus support our claim), a combinatorial model is developed that can be used to obtain a closed analytical formula for the number of I/O operations required.

In Chapter 6, we contend that a parallel sorting facility can be the basic building block for a relational database machine. It is shown that parallel sorting can be

used to efficiently implement the relational join. We conclude by summarizing the contributions of this thesis and by indicating some directions for future research in the area of parallel algorithms for database management.

CHAPTER 2

TAXONOMY OF PARALLEL SORTING

Many parallel sorting algorithms exist, most of which require a very large number of processors. Despite the apparent disparity among these algorithms, we contend that the majority fall into one of two categories: the network sorting algorithms and the shared memory sorting algorithms. The network sorting algorithms were the first fast parallel sorting algorithms to be developed. They have inspired algorithms for multiprocessors that have been built, or will be built in the near future. The shared memory sorting algorithms are faster than the network algorithms, but they are based on a theoretical model of parallel computation and cannot be implemented with current technology.

Since the sorting problem has been studied extensively for a uniprocessor, it would seem that efficient parallel sorting algorithms could be obtained by parallelizing well-known serial algorithms. However, this approach did not lead to a major breakthrough in parallel sorting. Instead, the development of very fast parallel sorting methods originated with the discovery of new iterative merging rules [Batc68, Vali75] that are intrinsically

parallel rules.

This chapter is organized as follows. In Section 2.1, we show that parallelizing some serial sorting algorithms can be done, but leads only to simple and relatively slow parallel algorithms. Section 2.2 is devoted to the network sorting algorithms. First, various types of sorting networks are surveyed. In particular, we describe in detail several sorting networks that perform Batcher's bitonic sort. Then, bitonic sort algorithms for SIMD (Single Instruction Multiple Data stream) computers such as the Illiac-IV are presented.

Section 2.3 surveys a chain of results that led to the development of very fast sorting algorithms: the shared memory model parallel merging [Vali75, Gavr75] and sorting algorithms [Hirs78, Prep78]. In Section 2.4, we complete our survey of the parallel sorting literature by describing Even's tape sorting algorithm [Eve74], which is the only known parallel file sorting algorithm. Finally, in Section 2.5 we briefly summarize our survey, and point out that several problems in parallel sorting require further investigation.

2.1. Parallelizing serial sorting algorithms

Parallel processing makes it possible to perform more than a single comparison during each time unit. Some models

of parallel computation (the sorting networks in particular) assume that a key is compared to only one other key during a time unit. This is restrictive but does not really limit the amount of parallelism because, in general, there are fewer processors available than pairs to compare. Another possibility is to compare a key to many other keys simultaneously. For example, in [Mull75], a key is compared to $n-1$ other keys in a single time unit using $(n-1)$ processors.

Parallelism may also be exploited to move many keys simultaneously. After a parallel comparison step, processors conditionally exchange data. The concurrency that can be achieved in the exchange steps is limited either by the interconnection scheme between the processors (if one exists), or by memory conflicts (if shared memory is used for communication).

The analog to a comparison and move in a uniprocessor memory becomes a parallel comparison-exchange of pairs with this parallel scheme. Therefore, it is natural to measure the performance of parallel sorting algorithms by the number of comparison-exchanges they require. Then the speedup of a parallel sorting algorithm may be defined as the ratio between the number of comparison-moves required by an optimal serial sorting algorithm and the number of comparison-exchanges required by the parallel algorithm.

Since an optimal serial algorithm sorts n keys in $O(n \log n)$ time, the optimal speedup would be achieved when n keys are sorted with n processors in time $O(\log n)$. However, it does not seem possible to achieve this bound by simply parallelizing one of the well known optimal serial sorting algorithms, since it appears that the best serial sorting algorithms have severe serial constraints that cannot be removed. On the other hand, parallelization of straight sorting methods (i.e. brute force methods requiring $O(n^2)$ comparisons) seems easier but it cannot lead to very fast parallel algorithms. By performing n comparisons instead of 1 in a single time unit, the execution time can be reduced from $O(n^2)$ to $O(n)$. An example of this kind of parallelization is a well known parallel version of the common bubble-sort, called the odd-even transposition sort (Section 2.1.1).

Partial parallelization of a fast serial algorithm can also lead to a parallel algorithm of order $O(n)$. For example, the serial tree selection algorithm can clearly be modified so that all the comparisons at the same level of the tree are performed in parallel. The result is a parallel tree sort, that is described in Section 2.1.2. This simple algorithm is used in the database Tree Machine [Bent79].

2.1.1. The odd-even transposition sort

The serial "bubble sort" proceeds by comparing and exchanging pairs of adjacent items. To sort an array (x_1, x_2, \dots, x_n) , $n-1$ comparison-exchanges $(x_1, x_2), (x_2, x_3), \dots, (x_{n-1}, x_n)$ are performed. This results in placing the maximum at the right end of the array. After this first step, x_n is discarded and the same "bubble" sequence of comparison-exchanges is applied to the array (x_1, \dots, x_{n-1}) . By iterating $n-1$ times, the entire sequence is sorted.

The serial odd-even transposition sort [Knut73, p.65] is a variation of the basic bubble sort, where comparisons are alternatively performed between odd elements and their right adjacent neighbor, or even elements and their right adjacent neighbor. There are n phases of comparison-exchanges. During odd phases, the pairs for the comparison-exchanges are $(x_1, x_2), (x_3, x_4), \dots$. During even phases, $(x_2, x_3), (x_4, x_5), \dots$ are compared and exchanged. To completely sort the sequence, it is possible to show that a total of n phases is required [Knut73, p.65].

This algorithm calls for a straightforward parallelization [Baud78]. Consider n linearly connected processors and label them P_1, P_2, \dots, P_n . We assume that the links are bidirectional, so that P_i can communicate with P_{i-1} and P_{i+1} . Initially x_i resides in P_i for $i=1, 2, \dots, n$. For a

parallel sort, let P_1, P_3, \dots be active during the odd time steps, and execute in parallel the odd phases of the serial odd-even transposition sort. Similarly, let P_2, P_4, \dots be active during the even time steps, and perform in parallel the even phases. Note that a single comparison-exchange requires 2 transfers. For example, during the first step, x_2 is transferred to P_1 and compared to x_1 by P_1 . Then, if $x_1 > x_2$, x_1 is transferred to P_2 ; otherwise, x_2 is transferred back to P_2 . Therefore, the parallel odd-even transposition algorithm sorts n numbers with n processors in n comparisons and $2n$ transfers.

2.1.2. A parallel tree-sort algorithm

In a serial tree selection sort, n numbers are sorted using a binary tree data structure. The tree has n leaves, and initially, each number is stored at each leaf. Sorting is performed by selecting the minimum of the n numbers, then the minimum of the remaining $n-1$ numbers, etc... The binary tree structure is used to find the maximum by iteratively comparing the numbers in two sibling nodes, and moving the smaller number to the parent node (see Figure 1). By simultaneously performing all the comparisons at the same level of the binary tree, a parallel tree-sort is obtained [Bent79]. If a processor is assigned to each of the $2n-1$ nodes of the tree, the minimum can be transferred

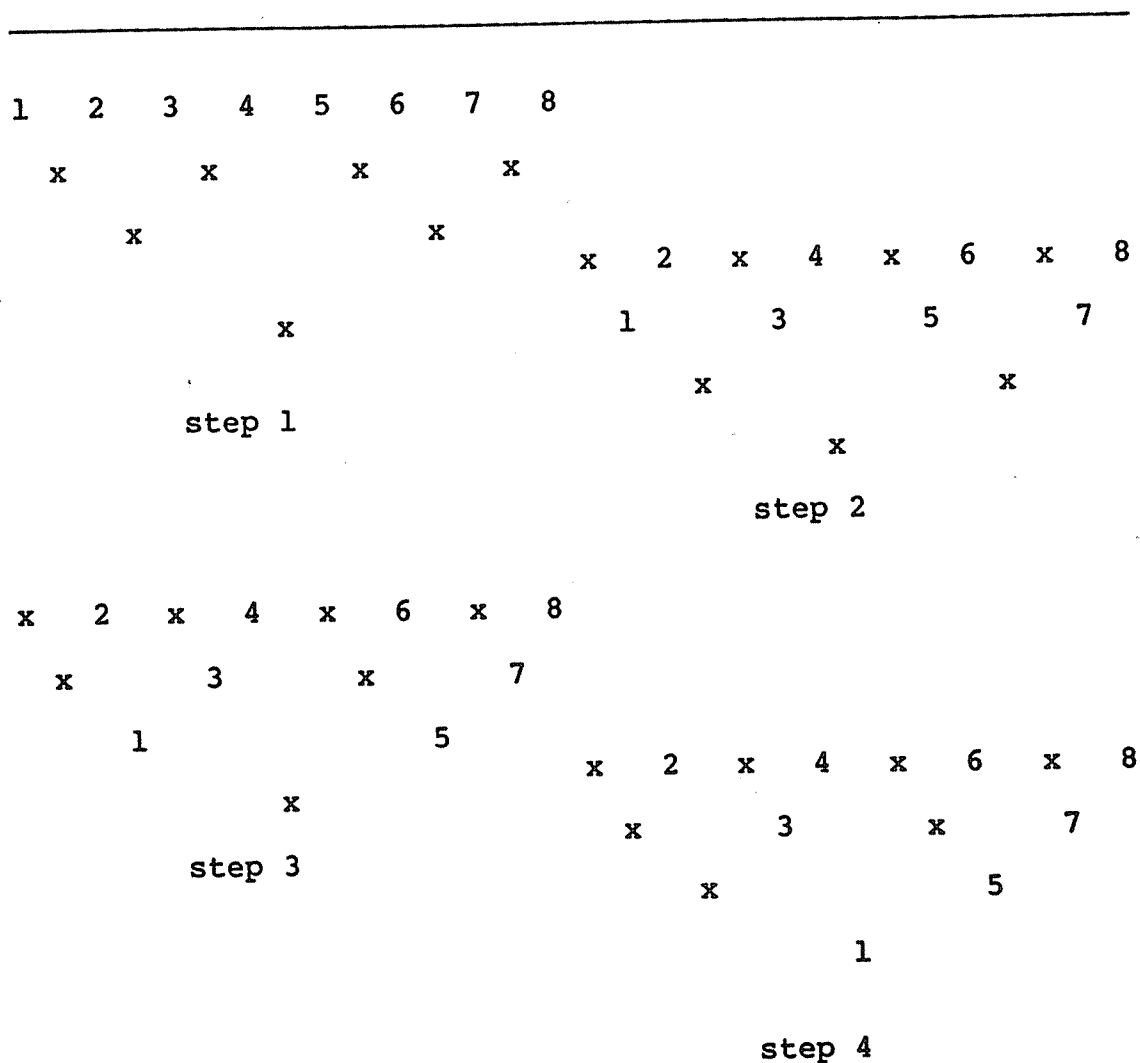


Figure 1. Tree selection sort

to the root processor in $\log_2 n$ comparison-transfer steps (assuming that a binary tree interconnection exists between the processors). At each step, a parent receives from its two children an element, performs a comparison, keeps the

smaller element and sends back the greater one. After the minimum has reached the root, it is written out. From then on, the processors are instructed to accept data from non-empty children, and to select the minimum, when they receive 2 elements. At every other step, the next element in increasing order reaches the root. Thus, sorting is completed in time $O(n)$.

In a similar manner, the serial heap sort can be parallelized, and the result is a parallel sort algorithm that sorts n numbers with n processors. But the execution time of this algorithm is also $O(n)$, due to the necessity to read off the sorted sequence serially at the root of the tree.

The speedup achieved with these simple parallelization schemes ($\log n$ for n processors) is not satisfactory and many efforts have been made to improve it. The first major improvement was reached by the sorting networks, that sort n numbers in time $(\log n)^2$ and thus, achieve a speedup of $n/\log n$ [Batc68]. Later, Preparata [Prep78] showed that the optimal bound (time: $O(\log n)$, speedup: n) can be achieved by using a theoretical model of n processors accessing a large shared memory.

Another important issue is whether the performance criteria by which parallel sorting algorithms have been

previously evaluated are general enough. Clearly, assuming that the number of processors is as large as the number of elements to be sorted and counting the number of parallel comparisons as the main cost factor is not satisfactory. Communication costs and, in the case of external sorting, I/O costs must be incorporated in a comprehensive analytical model, general enough to accommodate a wide range of multiprocessor architectures.

2.2. The network sorting algorithms

It is somehow surprising that a simple hardware problem, namely designing a multiple-input multiple-output switching network, has motivated the development and the proliferation of parallel sorting algorithms. The earliest results in parallel sorting are found in the literature on sorting networks [Voor71, Batc68]. In Section 2.2.1, we describe two types of sorting networks, respectively based on the odd-even and bitonic merge rules. In Section 2.2.2, we show that parallel sorting algorithms for SIMD machines can be derived from the bitonic network sort. In particular, we describe two bitonic sort algorithms for a mesh-connected processor [Thom77, Nass79].

2.2.1. Sorting networks

Sorting networks originated as fast and economical switching networks. Since it can order any permutation of

$(1, 2, \dots, n)$, a sorting network with n input lines can be used as a multiple-input multiple-output switching network [Batc68]. To realize a fast sorting network, it is necessary to exploit the possibility of performing comparisons in parallel, that can be provided by using a number of comparator modules. Implementing a serial sorting algorithm on a network of comparators results in a serialization of the comparators, and consequently, an increase in the network delay.

One of the earliest results in parallel sorting is due to Batcher, who exhibited two methods to sort n keys with $O(n \log^2 n)$ comparators in time $O(\log^2 n)$. A comparator is a module which receives two numbers on its two input lines A , B and outputs the minimum on its higher output line L and the maximum on its lower output line H (Figure 2). A serial comparator receives A and B with their most significant bit first and can be realized with a small number of NOR gates. Parallel comparators, where several bits are compared in parallel at each step, are faster but obviously more complex. Both of Batcher's algorithms, the "odd-even merge" and the "bitonic sort", are based on the principle of iterated merging. Starting with an initial sequence of 2^k numbers, a specific iterative rule is used to create sorted runs of length $2, 4, 8, \dots, 2^k$ during successive stages of the algorithm.

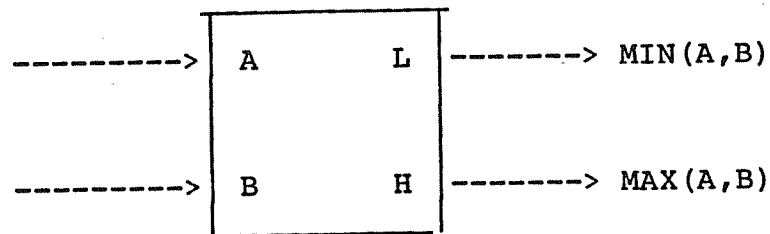


Figure 2. A comparison-exchange module

The odd-even merge rule:

The iterative rule for the odd-even merge is illustrated in Figure 3. Given two sorted sequences, (a_1, a_2, \dots) and (b_1, b_2, \dots) , two new sequences ("odd" and "even" sequences) are created: One consists of the odd numbered terms and the other of the even numbered terms from both sequences. The odd sequence (c_1, c_2, \dots) is obtained by merging the odd terms (a_1, a_3, \dots) with the odd terms (b_1, b_3, \dots) . Similarly, the even sequence (d_1, d_2, \dots) is obtained by merging the even terms (a_2, a_4, \dots) with the even terms (b_2, b_4, \dots) . Finally the sequences (c_1, c_2, \dots) and (d_1, d_2, \dots) are merged into (e_1, e_2, \dots) by applying the following comparison-exchanges:

$$e_1 = c_1$$

$$e_{2i} = \max(c_{i+1}, d_i)$$

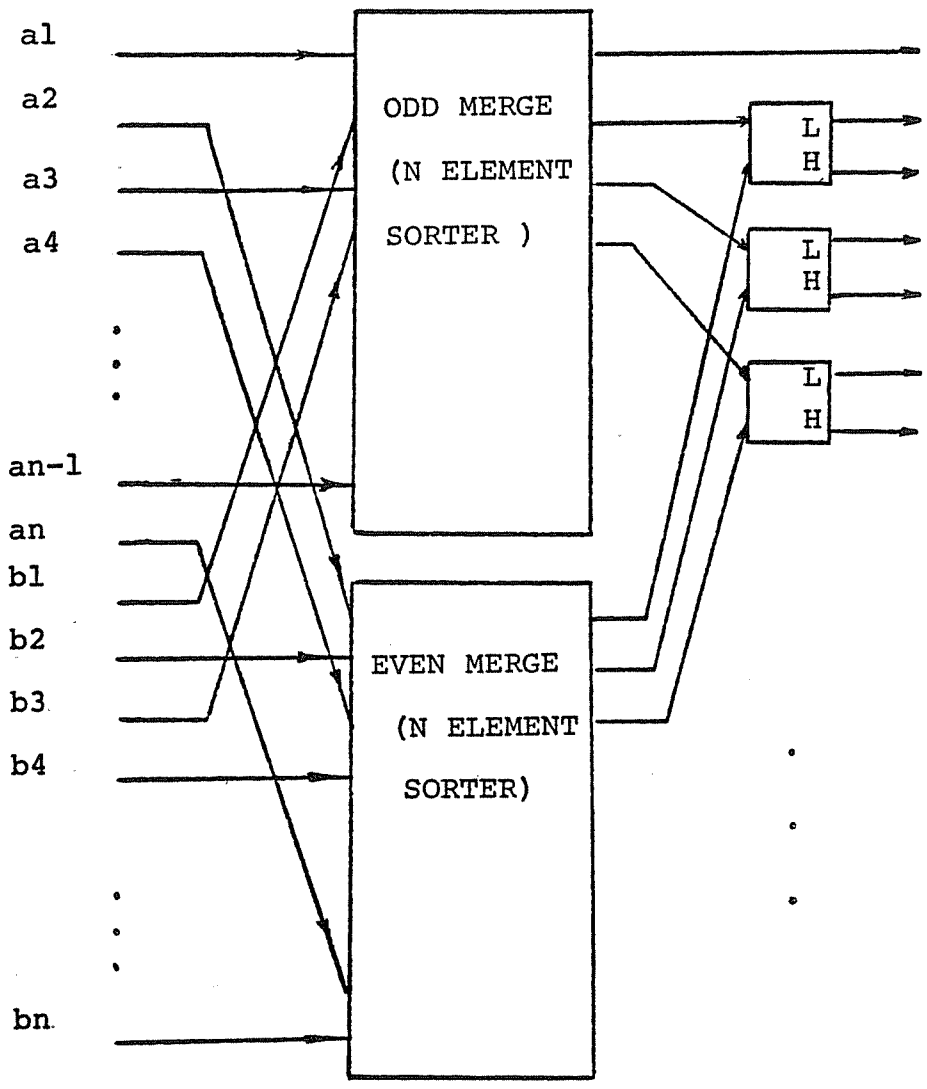


Figure 3. The odd-even merge rule

$$e_{2^{i+1}} = \min(c_{i+1}, d_i)$$

The resulting sequence will be sorted. (For a proof the reader is referred to [Knut73, p.224,225],

To sort 2^k numbers using the odd-even iterative merge requires 2^{k-1} 1 by 1 merging networks (i.e comparison-exchange modules), followed by 2^{k-2} 2 by 2 merging networks, followed by 2^{k-3} 4 by 4 merging networks, etc... Since a 2^{i+1} by 2^{i+1} merging network requires one more step of comparison-exchanges than a 2^i by 2^i merging network, it follows that an input number goes through at most $1+2+\dots+k=k(k+1)/2$ comparators. This means that 2 numbers are sorted by performing $k(k+1)/2$ parallel comparisons and exchanges. However, the number of comparators required by this type of sorting network is $(k^2-k+4)2^{k-2}-1$ [Batc68]. Several subsequent efforts [Knut73] have been able to reduce this number of comparators, but only for particular cases (e.g. $k \leq 4$).

The Bitonic merge rule:

For the "bitonic" sort a second iterative rule is used (Figure 4). A bitonic sequence is obtained by concatenating two monotonic sequences, one ascending and the other descending[1]. Examples of bitonic sequences are:

[1] A more general definition of a bitonic sequence allows a cyclic shift of such a sequence, e.g. 89642135 would also be a bitonic sequence.

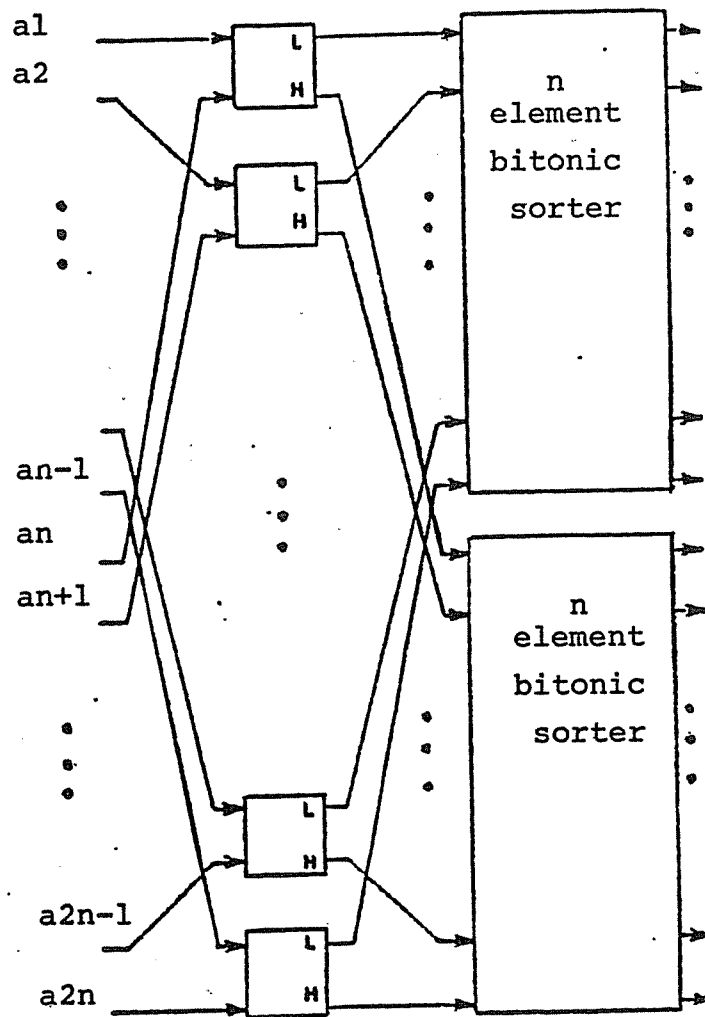


Figure 4. The Bitonic merge rule

3 5 8 9 6 4 2 1
4 7 9 6 5 4 3 2

The bitonic iterative rule is based on the observation that a bitonic sequence may be split into two bitonic subsequences by performing a single step of comparison-exchanges. Let $(a_1, a_2, \dots, a_{2n})$ be a bitonic sequence such that $a_1 \leq a_2, \dots, \leq a_n$ and $a_{n+1} \geq a_{n+2} \geq \dots \geq a_{2n}$. Then the sequences

$$\min(a_1, a_{n+1}), \min(a_2, a_{n+2}), \dots$$

and

$$\max(a_1, a_{n+1}), \max(a_2, a_{n+2}), \dots$$

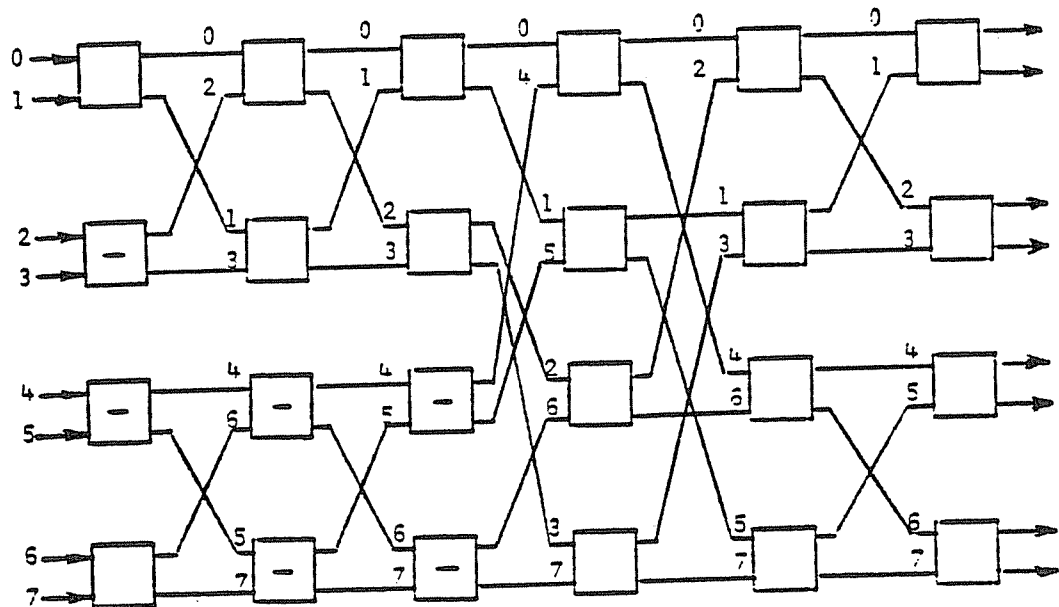
are both bitonic. Furthermore, the first sequence contains the n lowest elements of the original sequence while the second contains the n highest. It follows that a bitonic sequence can be sorted by sorting separately two bitonic sequences that are one half as long.

To sort 2^k numbers using the bitonic iterative rule we can successively sort and merge sequences into larger sequences until a bitonic sequence of size 2^k is obtained. This sequence can be split into a "lower" and a "higher" bitonic subsequences. Note that the recursive building procedure of a bitonic sequence must use some comparators that invert their output lines and output a pair of numbers in decreasing order. This is necessary in order to build the decreasing subsequence of a bitonic sequence (see

Figure 5). A bitonic sort of 2^k numbers requires $k(k+1)/2$ steps, each using 2^{k-1} comparators.

Since the first version of the bitonic sort was presented, the algorithm has been considerably improved by the introduction of the "perfect shuffle" interconnection [Ston71]. Stone noticed that if the inputs were labeled by a binary index, then the indices of every pair of keys that enter a comparator, at any step of the bitonic sort network, would differ by a single bit in their binary representations. Stone also made the following observations: The network has $\log_2 n$ stages. The i th stage consists of i steps, and at step i , inputs that differ in their i th least significant bits are compared. This regularity in the bitonic sorter suggested that a more regular interconnection could be used between the comparators of two adjacent columns of the network.

Stone concluded that the perfect shuffle interconnection could be used throughout all the stages of the network. "Shuffling" the input lines (in a manner similar to shuffling a deck of cards) is equivalent to shift their binary representation to the left. Shuffling twice shifts the binary representation of each index twice. Thus, the input lines can be ordered before each step of comparison-exchanges by shuffling them as many times as required by the bitonic sort algorithm. The network that realizes this




 : Indicates a comparator which inverts its output lines

Figure 5. Batcher's bitonic sort for 8 elements

idea is illustrated in Figure 6 for 16 input lines. In general, for $n=2^k$ input lines, this type of bitonic sorter requires a total of $(n/2)(\log n)^2$ comparators, arranged in $(\log n)^2$ ranks of $n/2$ comparators each. The network has $\log n$ stages, with each stage consisting of $\log n$ steps. At each step, the output lines are shuffled before they enter the next column of comparators. The comparators in the first $(\log n)-i$ steps of the i th stage do not exchange their inputs. Their only use is to shuffle their input lines.

Instead of a multistage network, the bitonic sort can also be implemented on a recirculating network, that requires a much smaller number of comparators. For example, an alternative bitonic sorter can be built with a single rank of comparators connected by a set of shift registers and shuffle links as shown in Figure 7.

Since the i th stage of the bitonic sort algorithm requires i comparison-exchanges, Batcher's sort requires

$$1+2+3+\dots+\log n = \log n(\log n + 1)/2$$

parallel comparison-exchanges. Stone's bitonic sorter, on the other hand, requires a total of $(\log n)^2$ steps, because additional steps are needed for shuffling the input lines (without performing a comparison). In both cases, the asymptotic complexity is $O(\log^2 n)$ comparison-exchanges. This provides a speedup of $O(\log n/n)$ over the $O(n \log n)$

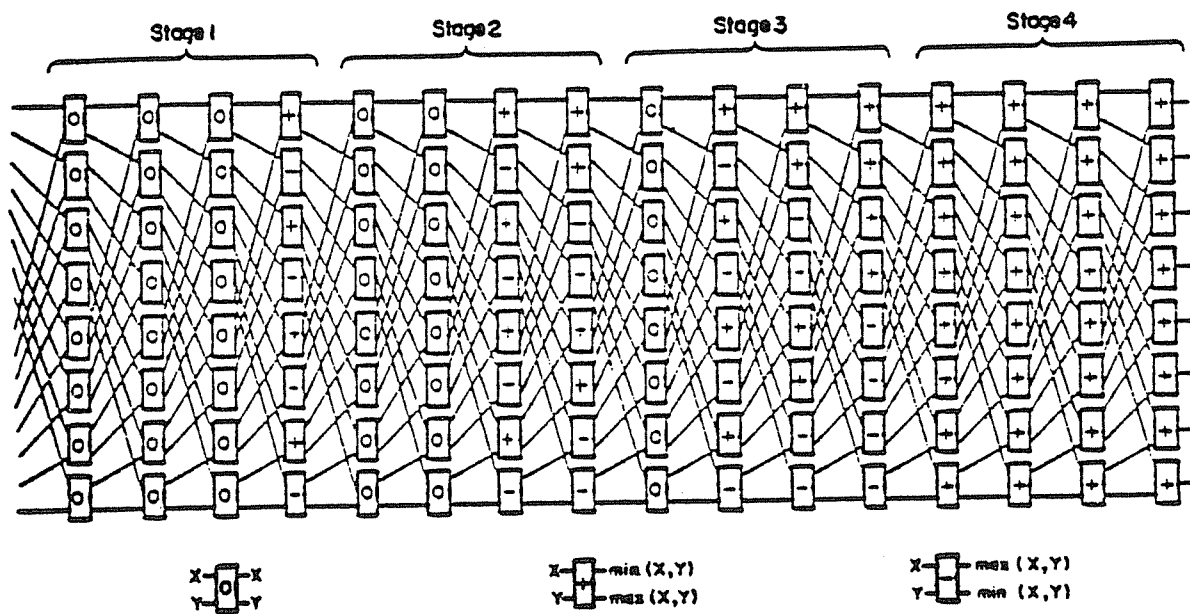


Figure 6. Stone's modified bitonic sort

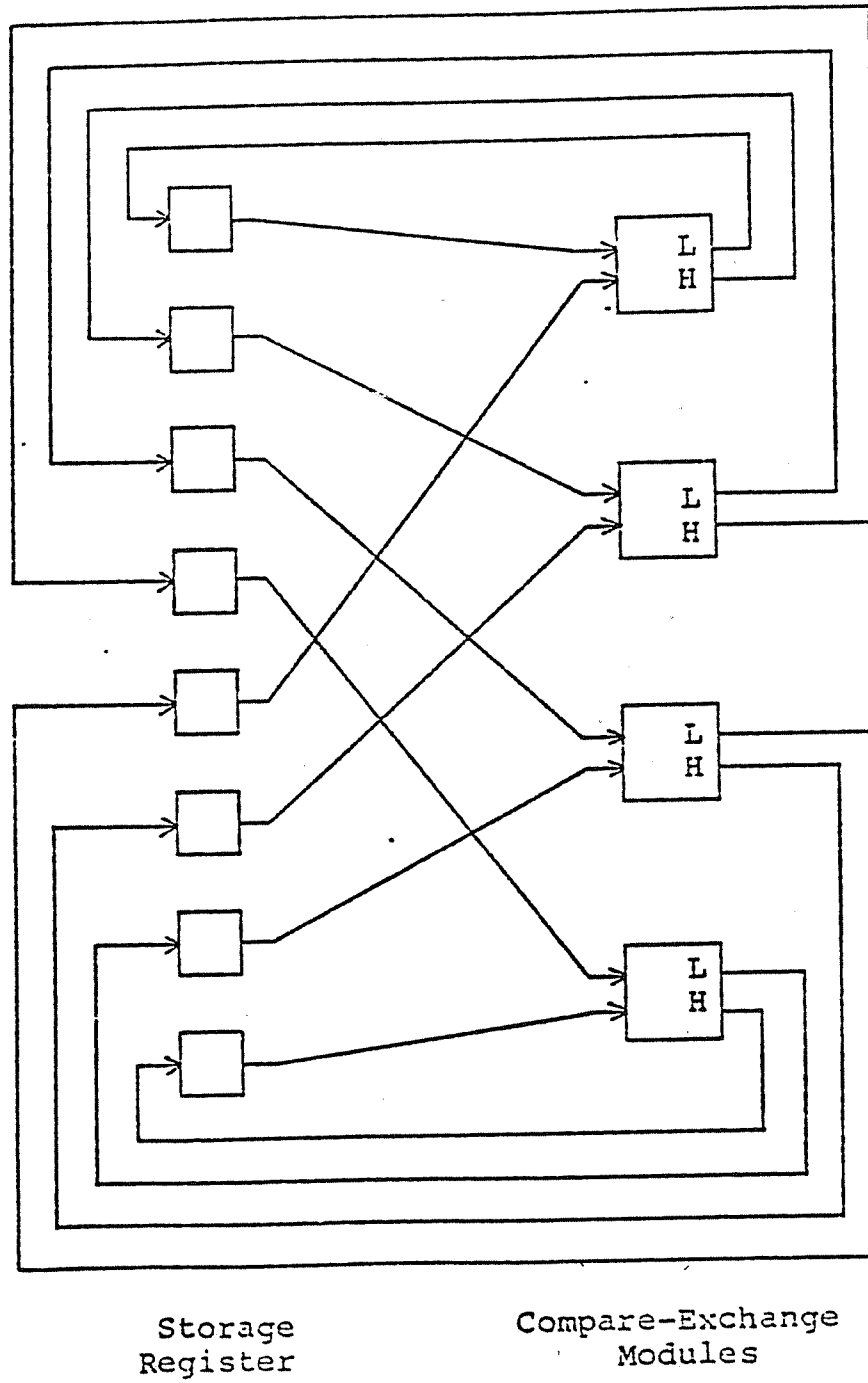


Figure 7. Stone's architecture for the bitonic sort

complexity of serial sorting. Therefore, it improves significantly the previous known bound of $O(n)$ for parallel speedup with n processors.

Siegel [Sieg77] has shown that the bitonic sort can be also performed by other types of networks in time $O(\log^2 n)$. Among the networks he considered, are the Cube and the Plus-Minus 2^i networks. Essentially, these networks can sort because they are able to simulate the perfect shuffle interconnection. Siegel proves that the simulation takes $O(\log^2 n)$ time, and thus, that sorting can also be performed within this time limit.

2.2.2. Sorting on an SIMD machine

Sorting networks are characterized by their "non adaptivity" property. They perform the same sequence of comparisons regardless of the result of intermediate comparisons. In other words, whenever two keys R_i and R_j are compared, the subsequent comparisons for R_i in the case that $R_i < R_j$ are the same as the comparisons that R_j would have entered in the case $R_j < R_i$. The non-adaptivity property makes the implementation of an algorithm very convenient for an SIMD machine. In particular, the sequence of comparisons and transfers to be executed by all the processors is determined when the sorting operation is initialized. Thus, a central controller can supervise the execution by

broadcasting at each time step the appropriate compare-exchange instruction to the processors.

Sorting on an array processor:

A different sorting problem is considered in [Thom75], in which the processors of an n by n mesh-connected multiprocessor are indexed according to a prespecified rule. The indexing rules considered are the row-major, the snake-like row-major, and the shuffled row-major rules (shown in Figure 8). Assuming that n^2 keys with arbitrary values are initially distributed so that exactly one key resides in each processor, the sorting problem consists of moving the i th smallest key to the processor indexed by i , for $i=1\dots n^2$. As with the sorting networks, parallelism is used to simultaneously compare pairs of keys, and a key is compared to only one other key at any given unit of time. Concurrent data movement is allowed but only in the same direction, that is all processors can simultaneously transfer the content of their transfer register to their right, left, above or below neighbor. This computation model is SIMD since at each time unit a single instruction (compare or move) can be broadcast for concurrent execution by the set of processors specified in the instruction. The complexity of a method which solves the sorting problem for this model can be measured in terms of the number of comparisons and unit-distance routing steps. For the rest

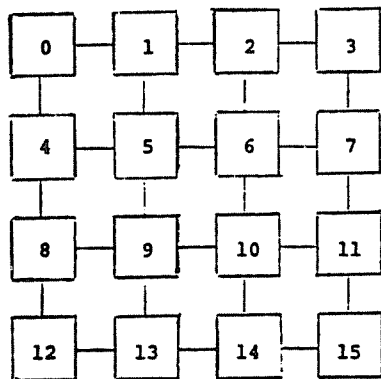
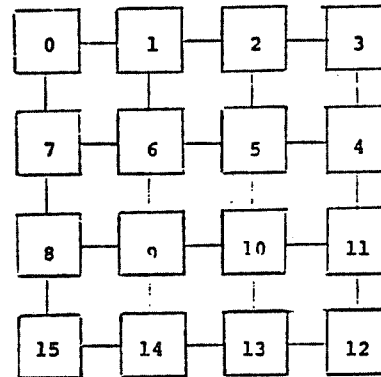
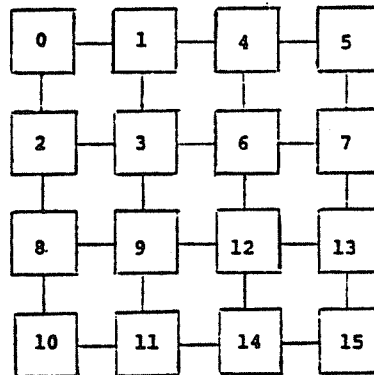
Row-major indexingSnake-like row-major indexingShuffled row-major indexing

Figure 8. Indexing schemes for an array processor

of this section we refer to the unit-distance routing step as a move. Any algorithm that is able to perform such a permutation will require at least $4(n-1)$ moves, since it may have to interchange the elements from two opposite corners of the array processor. This is true for any indexing scheme. In this sense a sorting algorithm which requires $O(n)$ moves is optimal.

In [Thom75], two algorithms are presented that perform this array sort in $O(n)$ comparisons and moves. The first algorithm uses an odd-even merge of two dimensional arrays and orders the keys with snake-like row-major indexing. The second uses a bitonic sort and orders the keys with shuffled row-major indexing. Recently, a third algorithm that sorts with row-major indexing with similar performance has been published [Nass79]. This algorithm is also an adaptation of the bitonic sort where the iterative rule is a merge of two dimensional arrays.

2.3. A shared memory model

After the bound of $O(\log^2 n)$ was achieved through the use of sorting networks, considerable effort was devoted to improve this result and to achieve the theoretical bound of $O(\log n)$. The first model that was able to reach this bound may be designated as a "modified" sorting network [Mull75]. Instead of comparison-exchange modules, this model uses

comparators which input two numbers A , B and output a single bit x : $x=0$ if $A < B$ or $x=1$ if $A > B$. To sort a sequence of n elements, each element is simultaneously compared to all the others by using a total of $n(n-1)$ comparators. The output bits from the comparators are then fed into a parallel counter which computes in $\log n$ steps the rank of a key by counting the number of bits set to 1 in the comparison of this key with all the other $n-1$ keys. Finally a switching network consisting of a binary tree of $\log n + 1$ levels of single-pole, double-throw switches routes a key of rank equal to i to the i th terminal of the tree. There is one such tree for each key, and each tree uses $2n-1$ switches. Routing a key through this tree requires $\log n$ time units, and this step determines the algorithm complexity. A diagram for this type of sorting network is presented in Figure 9.

At the cost of additional hardware complexity (the basic modules are more complex than comparison-exchange modules and the network uses more of them), the above algorithm sorts n keys in $O(\log n)$ time with $O(n^2)$ processing elements. This result was the first to use an enumeration scheme for parallel sorting. Later algorithms which we refer to as "enumeration type" sorting algorithms exploit the same idea.

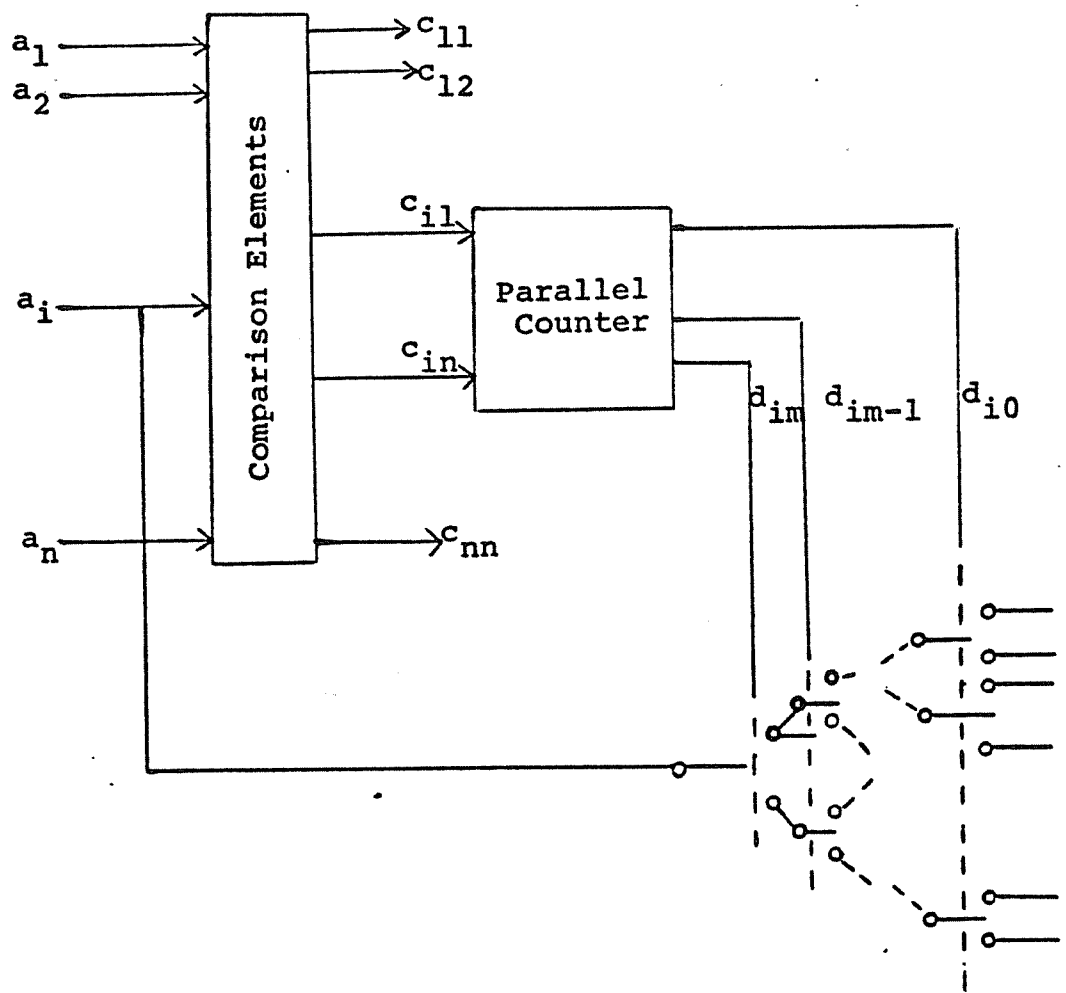


Figure 9. Muller's sorting network

The sorting network model and its modified version were embedded in a more general multiprocessor model where n processors have access to a large shared memory. With this scheme, sorting is performed by computing in parallel the rank of each key (the "enumeration" phase) and routing the keys to the location specified by their rank. The algorithm we have just described performs the enumeration with $n(n-1)$ comparators and the routing with n trees of $2n-1$ switches each. Therefore, it can also be described as an enumeration type algorithm which sorts n keys in time $O(\log n)$ on a multiprocessor which consists of $O(n^2)$ processors sharing a common memory of $O(n^2)$ cells.

A major improvement that this algorithm requires is a reduction in the number of processors. Even from a pure theoretical point of view, n^2 processors is a discouraging requirement for achieving a speed of $O(\log n)$. An optimal parallel sorting algorithm should achieve the same speed with only $O(n)$ processors in order to show a speedup of order n .

Faster parallel merge methods:

A shared memory model is also assumed in a study of parallelism in comparison problems by Valiant [Vali75]. Valiant's algorithm merges two sorted sequences of length n and m ($n \leq m$) with \sqrt{nm} processors in $2\log\log n + O(1)$ comparison

steps (compared to $\log^2 n$ for the bitonic merge). Another fast merging algorithm was developed by Gavril [Gavr75]. This algorithm merges two sorted sequences of length n and m with a smaller number of processors p ($p \leq n \leq m$). By using a simple parallel binary insertion method, the algorithm performs the merge operation with $2\log(n+1) + 4n/p$ comparisons when $n=m$. These two fast merge procedures were the basis for subsequent parallel sorting algorithms [Hirs78, Prep78] of optimal complexity $O(\log n)$.

Optimal parallel sorting algorithms:

Hirschberg's algorithm [Hirs78] is a "bucket sort" which sorts n numbers with n processors in time $O(\log n)$, provided that the numbers to be sorted are in the range $\{0, 1, \dots, m-1\}$. A side effect of this algorithm is that duplicate numbers are eliminated. If memory conflicts were ignored, it would be sufficient to have m buckets and to assign one number to each processor. The processor that gets the i th number would be labeled P_i ; P_i would then place the value i in the appropriate bucket. For example, if P_3 had the number 5, it would place the value 3 in bucket number 5. The problem with this simplistic solution is that a memory conflict may result when several processors attempt simultaneously to store different values of i in the same bucket. This memory contention problem may be solved by increasing substantially the memory requirements.

Suppose there is enough memory available for m arrays, each of size n . Each processor can then mark a bucket without any fear of memory conflict. To complete the enumeration sort the m arrays must be merged. This is done by using a sophisticated parallel merge procedure, where processors are granted simultaneous read access to a memory location but no write conflict can occur. Hirschberg generalizes the above method so that duplicate keys remain in the sorted array. But this degrades the performance of the sorting algorithm. The result is a method which sorts n numbers with $n^{1+1/k}$ processors in time $O(k \log n)$, where k is an arbitrary integer.

A major drawback of this algorithm (aside from the lack of realism of the shared memory model which will be discussed later) is its $m*n$ space requirement. Even when the range of possible values is not very large, one would like to reduce this requirement. In the case of a wide range of values (for example if the keys are character strings rather than integer numbers), the algorithm would not be applicable.

In [Prep78] two new fast enumeration-type sorting algorithms are presented. However, rather than computing separately the rank of every single element, they first partition the source array into a number of subarrays, sort the subarrays and compute partial ranks by merging pairs of

subarrays. Finally, for each element the sum of its partial ranks is also computed in parallel. The first algorithm uses Valiant's merging procedure [Vali75] and sorts n numbers with $n \log n$ processors in time $O(\log n)$. The second algorithm uses Batcher's odd-even merge and sorts n numbers with $n^{1+1/k}$ processors in time $O(k \log n)$. The performance of the latter algorithm is similar to Hirschberg's algorithm, but it has the additional advantage of being free from memory contention. Recall that Hirschberg's model required simultaneous fetches from the shared memory, while Preparata's method does not (since each key participates in only one comparison at any given unit of time).

Despite the improvement achieved by eliminating memory conflicts, these algorithms are still not very realistic. Any model requiring at least as many processors as the number of keys to be sorted, all sharing a very large common memory, is not feasible with present or near term technology. However, the results achieved are of major theoretical importance and the methods used demonstrate the intrinsic parallel nature of certain sorting procedures. Furthermore, it seems that many of the basic ideas in these algorithms can inspire the design and implementation of realistic parallel sorting methods for multiprocessors. For example, in Section 4.6, we present a simple method for parallel sorting by enumeration, that can be implemented on

a backend multiprocessor. While the efficiency of this method relies on the assumption that a fast broadcast facility is available, no shared memory is required.

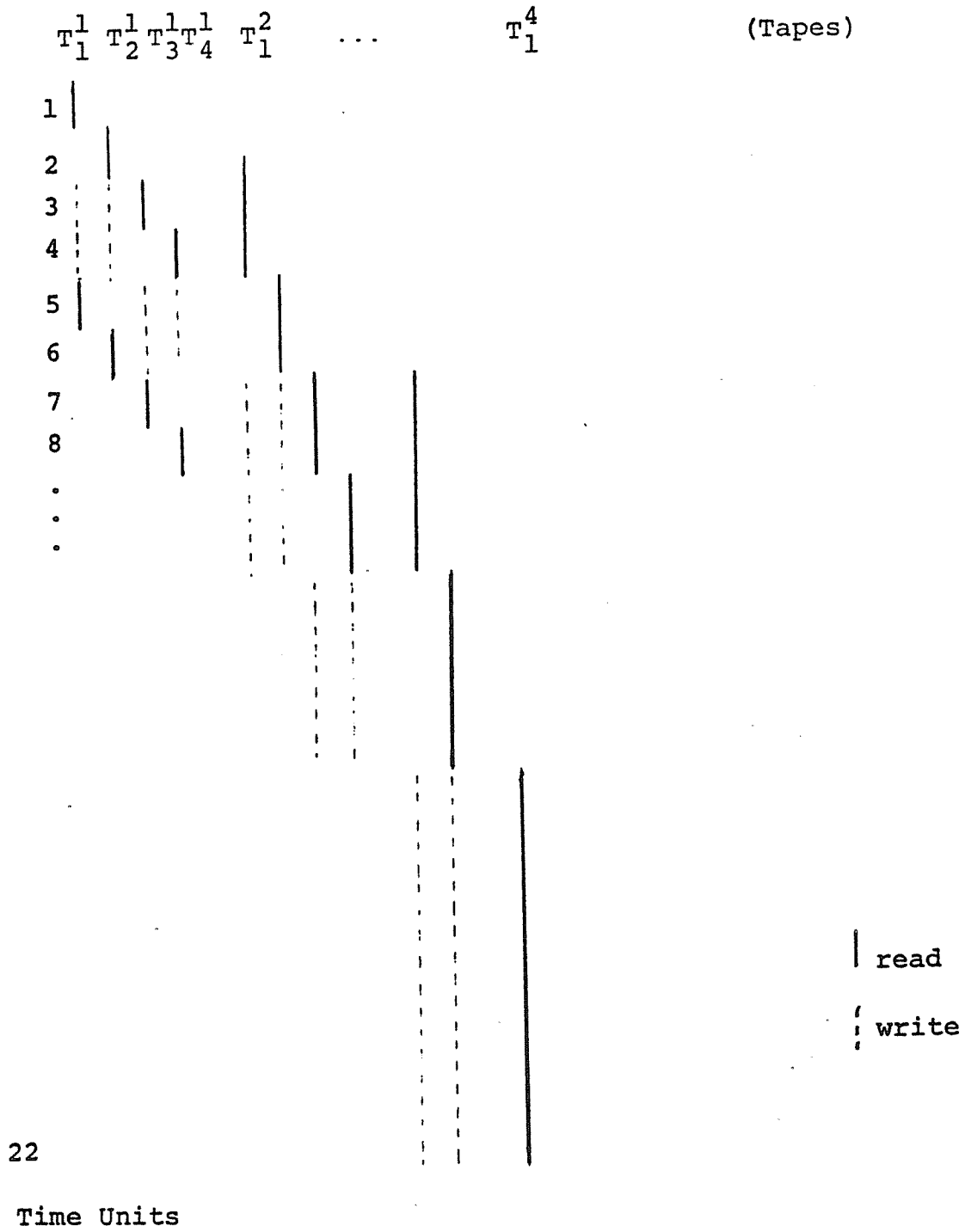
2.4. Parallel tape sorting algorithms

Each of the parallel sorting algorithms we have previously discussed can be classified as an internal sorting algorithm since they assume that all the data to be sorted resides in the processors' memory. However, in [Even74] two parallel tape sorting algorithms are presented that are external sorting algorithms. The sorting problem addressed in [Even74] is to sort a file of n records with p processors, where $p \ll n$. The only internal memory requirement is that three records could fit simultaneously in each processor's local memory. Both algorithms are parallel versions of an external 2-way merge-sort. They sort a file of n records by merging iteratively pairs of sorted runs of size $2, 2^2, \dots, 2^{\lceil \log n \rceil - 1}$. In the first method each processor is assigned n/p records and 4 tapes to perform an external merge sort on this subset. After p sorted runs have been produced by this parallel phase, a single processor merge-sorts them serially.

In the second method, the basic idea is that each processor performs a different phase of the serial merge procedure. The i th processor merges pairs of runs of size

2^{i-1} into runs of size 2^i for $i=1, 2, \dots, \lceil \log n \rceil$ (ideally n is a power of 2 and $\log n$ processors are available). A high degree of parallelism is achieved by using the output tapes of a processor as input tapes for the next processor so that as soon as a processor has written 2 runs these runs can be read and merged by another processor. In order to overlap the output time of a processor with the input time of its successor, each processor writes alternately on 4 tapes (one output run on each tape). This procedure also reduces the time for rewinding the tapes, since every tape is emptied (that is the records it contains are read) before it is loaded again with a sorted run (Figure 10).

The pipelined merge sort that we present in Section 4.2 is an adaptation of this tape sorting algorithm for our multiprocessor model. We show that the delay between processors can be shortened and that the 4p magnetic tapes requirement can be replaced by proper use of a modified moving-head disk.



22

Figure 10. Even's tape sorting algorithm

2.5. Summary

One conclusion emerges clearly from this survey: Most research on parallel sorting has concentrated on finding new ways to speedup the algorithms' theoretical computation time, while constraints other than time have received little consideration. Typically, algorithms have been developed that require n processors (and in some cases, even more than n) to sort n numbers. Figure 11 summarizes the number of processors and the computation time required by various algorithms described in this chapter.

2.5.1. Implementation and parallel architecture constraints

A general criticism against recent research in parallel sorting, is that it ignores most architectural constraints. As indicated in Figure 10, most algorithms require a very large number of processors. This requirement was initially justified, when parallel sorting algorithms were required for implementing efficient switching networks. In this context, the processors are simple hardware boxes, that only compare and exchange their two inputs. Also, the number of input lines to a switching network is never as prohibitive as the number of elements that a general purpose sorting algorithm may have to sort. However, when parallelism is invoked for speeding-up sort-

Algorithm	Number of processors	Execution time	Other characteristics
odd-even transp.	n	$O(n)$	
Batcher's bitonic	$n \log^2 n$	$O(\log^2 n)$	sorting network
Stone's bitonic	$n/2$	$O(\log^2 n)$	sorting network
Mesh bitonic	n^2	$O(n)$	sorts n^2 el.
Muller-Preparata	n^2	$O(\log n)$	
Hirschberg	n	$O(\log n)$	duplicates probl
Hirschberg	$n^{1+1/k}$	$O(k \log n)$	memory conflicts
Preparata 1.	$n \log n$	$O(\log n)$	
Preparata 2.	$n^{1+1/k}$	$O(k \log n)$	no memory confli

Figure 11. Processors required and computation time

ing of a large array and the use of full-scale processors is implied, then architectural considerations are at least as important as the theoretical optimality of an algorithm.

An experiment on the STAR computer [Ston78] demonstrated that the performance of a sorting algorithm can be

significantly affected by the architecture of the processor on which it is implemented. Stone observed that although Quicksort had a better complexity than Batcher's bitonic sort on the vector computer STAR ($O(n \log n)$ compared to $O(n \log^2 n)$), when both algorithms were implemented, Quicksort was actually much slower. Stone explained this result by showing that the constants involved in the asymptotic bounds were very different, because of the way the algorithms were coded in vector instructions.

For the parallel algorithms that we have classified as "network sorting algorithms", the interconnection topology is a major factor in an efficient implementation. On the other hand, for the shared memory sorting algorithms, a factor that was not accounted for is the cost of assigning and synchronizing the processors. Further research and implementation efforts will probably indicate that not only the theoretical complexity of an algorithm determines its performance. Thus, "optimality" of an algorithm may be limited to the context of a specific architecture.

As indicated by our survey, the shared memory model algorithms have the best asymptotic complexity. However, it is most unlikely that future technology will supply the tools for implementing any of these algorithms. It may be the case that some of the simpler algorithms that have been surveyed (such as the odd-even transposition sort, for

example), will be the best compromise between optimality and feasibility.

2.5.2. Further aspects of parallel sorting

One aspect that has not been investigated at all, is the way the performance of an algorithm may be affected by the initial distribution of the data. This question is irrelevant for sorting networks, because the algorithms based on sorting networks are non-adaptive (see Section 2.2). However, the other parallel sorting algorithms that have been discussed could perform differently when the initial data is partially sorted. It would certainly be desirable to have some indication for worst and average case behavior of a parallel sorting algorithm, but results of this kind can generally be achieved only through experiments. Many empirical results are available for commonly used serial sorting algorithms. Since most known parallel sorting algorithms cannot be implemented on current multiprocessors, it is unlikely that experimental runs, on real machines and real data, will be possible in the near future.

Another aspect of parallel sorting algorithms that needs further investigation is the time-space tradeoff. This tradeoff was discussed for serial sorting in Section 1.2. We pointed out that an algorithm that requires a sub-

stantial amount of work space in memory is often faster and simpler than other algorithms that can sort data in situ. But despite its advantages, it would be ruled out because of memory limitations. In a parallel processing environment, the size of main memory remains a major constraint, whether or not it is shared by multiple processors. However, designers of parallel sorting algorithms often discard this constraint, and put the emphasis on execution time rather than space efficiency. As indicated in Section 2.3, the fastest parallel algorithms require as much as $O(n^2)$ space to sort n elements [Hirs78].

One feature common to parallel sorting algorithms that have been described in this chapter, is that there is no mention of how the elements to be sorted have been read in the processors' memories. While it is justified to elude this issue when considering a serial, internal sorting algorithm, the situation is different with parallel processing. On a single processor, the source data can only be read sequentially into memory. But for a multiprocessor, there is the possibility that several processors can read or write simultaneously. On the Illiac-IV, for example, a fixed-head moving disk was used for concurrent I/O by all 64 processors. However, it is more reasonable to assume that when a significantly larger number of processors is involved, only some of them will be able to perform

I/O operations concurrently. Thus, for parallel array sorting, we conclude that the cost of reading and writing the data should be taken into account when an algorithm is evaluated. In particular, there would be no point in using a parallel sorting algorithm that requires only $O(\log n)$ time, if the startup cost to get the data in memory is $O(n)$.

CHAPTER 3

A MODEL FOR PARALLEL EXTERNAL SORTING

For a conventional computer system, the distinction between array sorting methods and file sorting methods is well known, and there are well accepted criteria to measure their respective performance. However, as indicated by our survey of parallel sorting in Chapter 2, the topic of parallel file sorting has not yet been investigated (except for one very particular result [EVEN74] on parallel tape sorting).

The motivation for designing parallel file sorting algorithms and for developing a cost analysis framework for these algorithms is especially strong in the database management area. In conventional database management systems (DBMS's), sorting is known to be a very efficient tool for performing complex operations. In particular, it has been shown [Blasg77, Astr76] that the relational join can be very efficiently realized by pre-sorting the operand relations. However, research on database machines has concentrated on parallel processing tools other than parallel sorting, to improve the performance of DBMS's. In particular, several designs of associative disks have been proposed, that allow a very fast execution of record searching

[Slot70, Ozka77, Bane79]. The first step in introducing parallel sorting as a tool to be used by future database machines is to develop fast methods for sorting large files in parallel.

In a previous paper [Bora80], we have proposed two parallel file sorting methods, and estimated their execution time if they were run on a specific database machine. The machine was essentially modeled as DIRECT [DeWi79], and used a data cache for interprocessor communication and disk access. Here, we depart from this shared memory model, and propose a more general approach to parallel file sorting.

In this chapter, a model for parallel external sorting is investigated. This model will be used in the next chapter to present several new parallel external sorting algorithms. We begin by stating explicitly the problem of parallel file sorting, and by defining some terminology. We then investigate several aspects of this problem that need to be modelled. More specifically, the chapter is organized as follows. In Section 3.1, the notions of logical order of records and physical order on a disk device are clarified. Then, the concept of (serial) external sorting is extended to multiprocessors, and a definition of parallel external sorting is proposed. In Section 3.2, we specify the parallel architecture model that we propose to

use. A major component of this architecture is a modified moving-head disk device, that allows parallel access to tracks on the same cylinder. Finally, Section 3.3 describes a cost model for evaluating the performance of external parallel sorting algorithms. The analysis of the algorithms presented in Chapter 4 will be based on this model.

3.1. Problem statement and terminology

3.1.1. Record and key sorting

Since we are addressing the problem of sorting a file of records, some assumptions about the file logical structure, and about the meaning of sorting should be clearly stated. We assume that the file is composed of fixed length records. Each record is composed of a number of attribute-value pairs. By sorting a set of records, we mean ordering them in increasing order, with respect to the values of one or several attributes. These attributes constitute the sort "key". The key may be only a few characters long (e.g. a name or an identification number for a file of employees), or it may be as long as the entire record. The concatenation of the key attributes values constitutes an alphanumeric string, and the sort order is defined as the ascending, lexicographic order of these values. When describing sorting algorithms, we will use

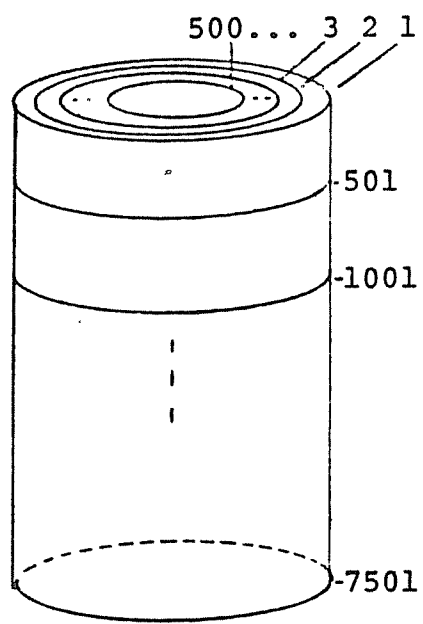
the terms "record sorting" and "record comparisons". Rather than specifying each time what the key is, we adopt the convention that record sorting means ordering records with respect to their key value, and comparing records means comparing their key values. However, when computing the execution time of an algorithm, we will consider the impact of varying the key length, since it may take significantly more time to compare very long keys than short ones. We do not assume that the key values are unique, even in the case that the key is the entire record (in this case, the file may contain duplicate records). Thus, when two records are compared, we may conclude either that one is "greater" than the other, or that they are equal.

3.1.2. Physical order on a disk

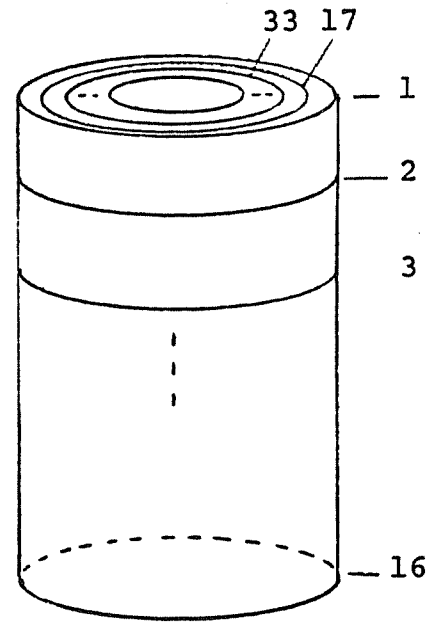
When records are stored on a sequential media such as magnetic tape, the meaning of physically ordered records is clear. However, when a magnetic disk device is used as the mass storage media, the concept of physical order must be defined. If a set of records fits on a single track, the physical order is determined, since a track is a sequential unit of storage. However, when the set is larger than a track, a convention is needed for ordering tracks. To gain intuition on the number of tracks that a large data file may occupy, suppose that the size of the file is about 10 Megabytes. Current disk devices have a track capacity

close to 20,000 bytes, and 15 to 30 tracks per cylinder (for example, the IBM 3350 disk has a 19,069 bytes track capacity and 30 tracks per cylinder). Thus, a 10 Mbyte file will occupy approximately 500 tracks. In allocating "contiguous" disk space to store the file, we may either decide to fill up contiguous cylinders, or to use contiguous tracks on a single disk surface. The consequent track numbering for these two choices is shown in Figure 12. In the first case, the file would span 17 cylinders on an IBM 3350 disk. In the second case, the file would occupy one surface.

If either of these two schemes is adopted for numbering tracks, then at the end of a sort operation, the algorithm must write the sorted file on 500 "contiguous" tracks. In addition, during intermediate stages of the sorting process, whenever a sorted subset of records is constituted, this subset must also be written onto contiguous tracks. For example, if a sorting algorithm is based on iterative merging, it will produce sorted runs of increasing size at each iteration. When a run becomes larger than a single track and requires two tracks for storage, it must be written on two neighboring tracks (that is one under the other on the same cylinder, or one next to each other on the same surface).



consecutive tracks on one surface



consecutive tracks on one cylinder

Figure 12. Physical Order on a Disk

Either disk layout scheme can have a significant impact on the time required for disk access during all stages of the sorting algorithm. Intuitively, it seems that compacting the runs on as few cylinders as possible should reduce the overall seek time. When sorting is performed by a parallel processor, the possibility of reading and writing from all tracks of the same cylinder simultaneously can also enforce this choice (assuming, of course, the availability of a parallel read/write disk). On the other hand, writing an entire run on a single surface may save communication and I/O time, when a processor is associated with each surface of the disk. In this case, a single processor cannot directly read or write a run that is stored on several tracks of a cylinder.

A third option is writing consecutive blocks of a sorted run on tracks allocated randomly, one at a time, by a controller. In this case, an address translation mechanism must be used to identify the sequence of tracks containing a sorted run. This approach, however, would make it almost impossible to minimize the disk access time of an algorithm. In addition, when a very large file is involved, the necessity of maintaining a large page table can also degrade significantly the performance of the I/O system. For a 100 Mbyte file, if there was one table entry for each disk block, the file directory would require as

much as 100K bytes. Because of its size, it may not be possible to maintain the entire table in the processors' memory. As a result, a single data fetch may cause two page faults (one for the page table, the other for the data page). Thus, even in the case of a conventional processor, the contiguous allocation of disk space for storing large files is advocated [Ston81].

3.1.3. Multiprocessor internal and external sorting

While the distinction between internal and external sorting algorithms for a uniprocessor has been discussed earlier, these concepts need to be redefined for a multiprocessor environment. A parallel algorithm is an internal sorting algorithm if it operates on data that resides in the multiprocessor's random-access memory; by sorting, we mean that the data is permuted within this memory so that, at termination of the algorithm, the data is arranged in a prespecified order. When this memory is shared by the processors, the data occupies a contiguous area of memory, and the order is clearly defined. On the other hand, when each processor has its own local memory, the notion of order must be defined. Usually, the interconnection topology suggests a natural ordering scheme. The processors are labelled by a serial number (e.g 1 to n), and sorting can be defined as bringing the smallest datum (or the first storage unit of data) to processor 1, the next to processor

2, ... , and the last to processor n.

A parallel sorting algorithm is defined as a parallel external sorting algorithm if it can sort a collection of records that is too large to fit in the total memory available in the multiprocessor. This definition is general enough to apply to both categories of architectures: the shared memory multiprocessors and the loosely coupled multiprocessors. For shared memory multiprocessors, an external sorting algorithm is required when the shared memory is not large enough to hold all the records (and some work space to execute the sort). On the other hand, for loosely coupled multiprocessors, the assumption is that the source records cannot be distributed among the processors' local memories. That is, the multiprocessor has p identical processors, and each processor's local memory is large enough to hold k records, but the source file has more than $p*k$ records. In both cases, the processors can access a mass storage device on which the file resides. At termination of the algorithm, the file must be written back to the mass storage device, in sorted order. This order must be defined according to the physical characteristics of the storage device. For a magnetic disk, we will use the physical order definition given at the beginning of this section.

3.2. A class of multiprocessors

The multiprocessor organization on which our parallel algorithms are based consists of the following components:

- (1) A set of general purpose processors.
- (2) An interconnection device connecting the processors.
- (3) A controlling processor
- (4) A modified moving head disk device, that allows for parallel read and write of tracks on the same cylinder.

A diagram showing this organization is shown in Figure 13, for 32 processors. In general, we assume that there are as many processors as disk heads, and that processors are physically associated with disk surfaces. Thus, once the disk heads are positioned on a cylinder, the processors can access in parallel all the tracks on the cylinder. Parallelism can be increased (above the limit imposed by the number of disk surfaces [1]), by using several disk drives, and by associating a set of processors with each drive.

[1] This number is typically between 15 and 30 for commercially available disks. For example, the IBM 3330 has 19 tracks per cylinder, the IBM 3350 has 30 and the IBM 3380 has 15.

3.2.1. Processor synchronization

Unlike the network internal sorting model (see Section 2.2), our model for external sorting does not rely on an SIMD machine architecture. Since external sorting requires the transfer of large amounts of data, we felt that the unit of data transfer should be as large as possible. Increasing the size of the data transfer unit reduces the communication overhead (since a message transfer has a fixed overhead cost, in addition to a cost per byte transferred). On the other hand, this size is clearly limited by the size of the processors' local memory. Our algorithms assume that data is transferred in page units, with a page size equal to the capacity of a typical disk track. Between two data transfers, the amount of computation executed by a processor can be significant. Thus, the granularity for processor synchronization intervals is of the order of several thousand instructions, rather than a single instruction.

Recent research on database machines also advocates an MIMD approach by showing its superiority over an SIMD approach for the support of inter-query concurrency [DeWi79], and for processor allocation strategies [Bora81]. Since we foresee that the main application of parallel file sorting algorithms will be in the database area, the architecture we propose is

influenced by these earlier results.

In our multiprocessor model, the processors operate independently of each other but can be synchronized by exchanging messages among themselves or with a controlling processor. At initiation time of a sorting algorithm, the controller assigns a number of processors to its execution. Because several other operations may be already in the process of being executed, the controller maintains a free list and assigns processors from this list. In addition to the availability of processors, the size of the sorting problem is also taken into consideration by the controller to determine the optimal processor allocation. Finally, some algorithms allow for a dynamic allocation of processors. That is, if they begin executing with less than the optimal number of processors, the controller can assign to them additional processors during intermediate stages of execution. The "pipelined selection sort" and the "pipelined merge sort" have this property. When these two algorithms are presented in Chapter 4, we will investigate in more detail the implications of a dynamic processor allocation strategy.

In addition to assigning processors, the controller is also responsible for coordinating and supervising the actions of the processors executing a task. For example, it maintains control tables that keep track of the current

files being read or written. Most sorting algorithms create intermediate sorted "runs" of data, that must be assigned to other processors for subsequent processing. In particular, in iterative merging procedures, a processor generates an output run that becomes an input run for another processor. In this case, the controller must maintain task tables that contain information about the source and destination processors for these runs.

3.2.2. The I/O device

A high degree of I/O bandwidth is critical for achieving a cost-effective performance in parallel file sorting. The cost of external sorting is dominated by the cost of I/O transfers. Therefore, if a parallel processor is to achieve any significant speedup, it must be able to read from and write to mass storage faster than a single processor. Ideally, one would have as many storage devices as processors, and a processor would read from and write to as many devices as are required by the sorting algorithm. This was essentially the approach taken in [Even74a], where 4 tape drives are assumed for each processor. However, a model of this kind, though it may help understand algorithmic aspects of parallel processing, does not take into consideration constraints imposed by technology. Like the shared memory model for array sorting, a parallel file sorting model that assumes a shared mass storage device

with infinite I/O bandwidth provides very limited insight into implementation aspects.

We have chosen to model our I/O device as a modified moving head disk. Disks that allow for parallel read and write of all the tracks on one cylinder have been proposed [Bane78.] and, in some cases already built [Leil78]. They appear to be a good compromise between the cost-effective, conventional moving-head disk and the obsolete fixed-head disk.

In order to minimize seek time, we propose to concurrently use at least two disk drives (see Figure 13). During execution of a single phase of a sorting algorithm, one drive can be utilized for reading and the other for writing. In Section 3.3, we demonstrate that having a separate drive for output makes it possible to concurrently write several output runs while minimizing seek time. In addition, a two-disk organization permits overlapping the write time with the computation and the read times.

Finally, we propose an alternative multiprocessor organization (see Figure 14). As we will demonstrate in Chapter 4, this organization can improve significantly the performance of parallel file sorting algorithms. The cost, however, is an increase in the complexity of the hardware. In the previous organization, a processor was physically

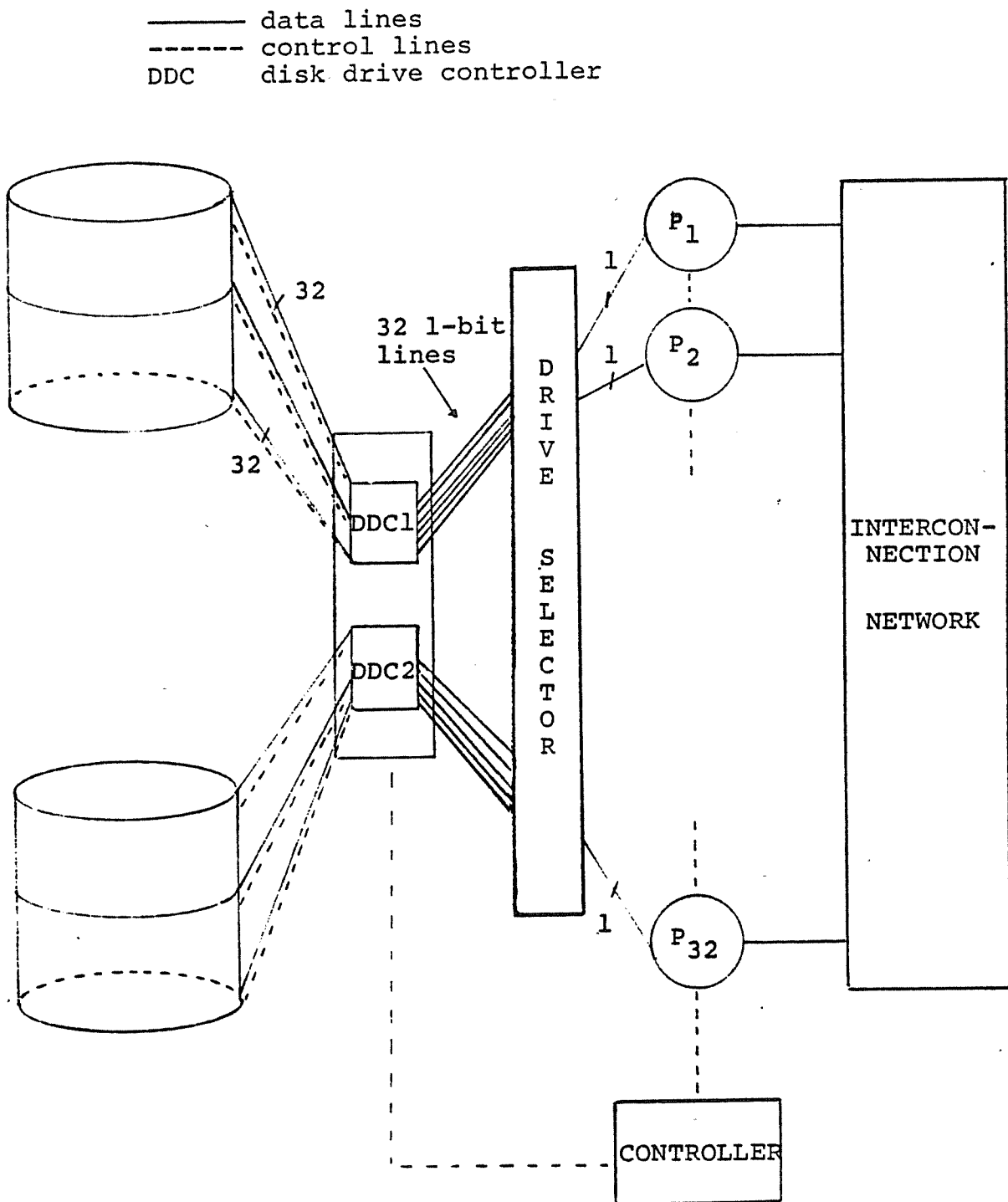


Figure 13. A Multiprocessor Architecture

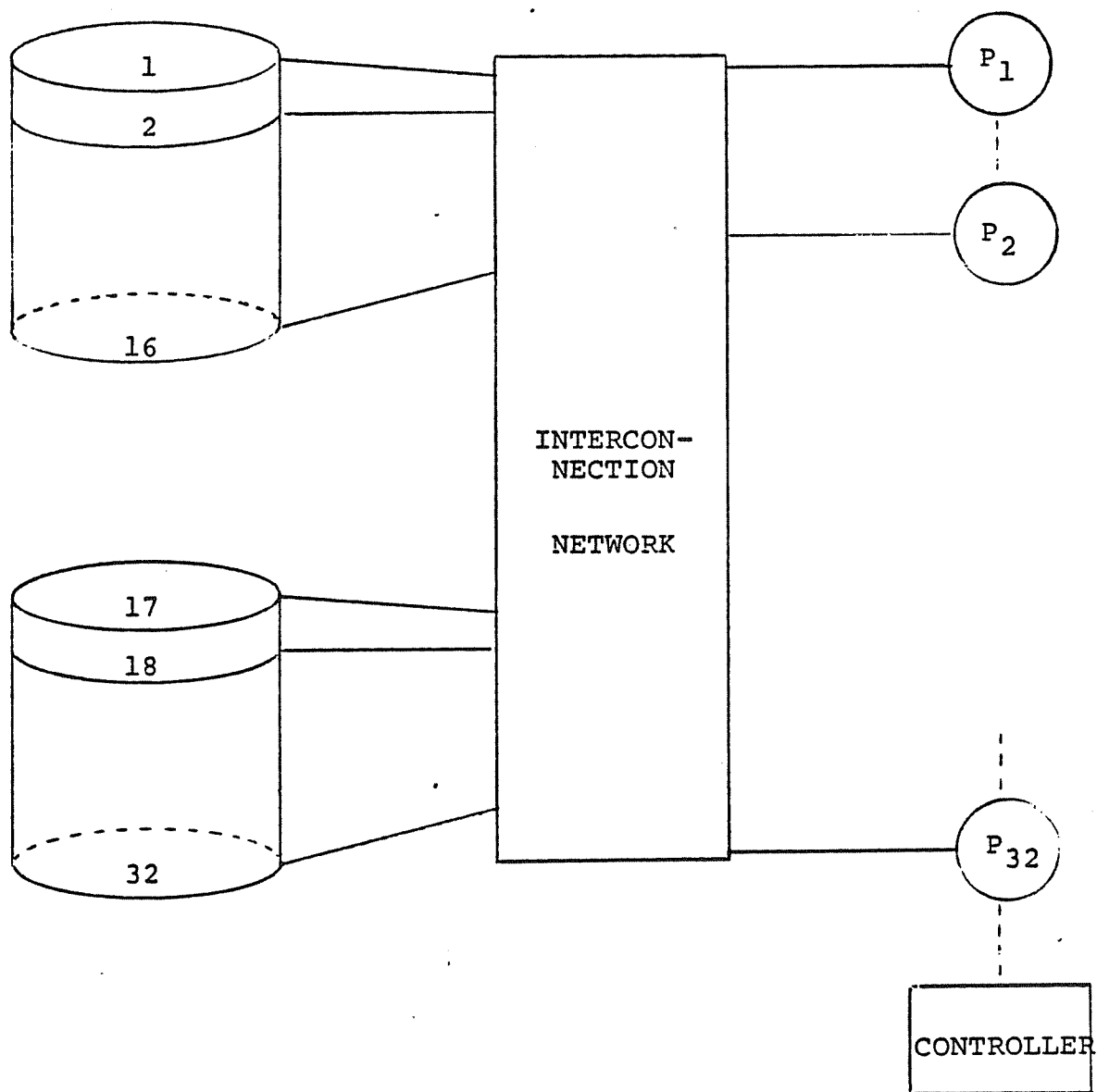


Figure 14. An Alternative Multiprocessor Architecture

associated with a specific surface of a disk drive. When we began to analyze several of the parallel sorting algorithms, we realized that numerous processor to processor and processor to mass storage transfers were incurred when it was necessary for a processor to access data from another disk surface. These transfers occur only because the processors are connected to the disk platters in a rigid manner. If the processors are instead connected to the disk heads by an adequate interconnection network, the number of interprocessor data transfers required for sorting can be significantly reduced.

3.3. A cost evaluation model

After some attempts to analyze the execution time of our algorithms, we felt the need for a rigorous definition of some basic performance parameters. These parameters must measure the I/O cost, the communication cost, and the processing cost for executing an algorithm on a given multiprocessor architecture. We have identified a number of basic tasks common to all our algorithms (e.g. reading a page from disk), and have associated a cost parameter with each. The execution-time expression for each algorithm will be expressed in terms of the costs of these basic steps. For different processors and mass storage device characteristics, the parameters may have different values and may relate differently to each other.

Our first basic assumption is that data is moved and processed by page units. The page size is determined by the capacity of a disk track, and by the size of the processors' local memory. Our analysis assumes that this memory capacity is approximately three times the size of a disk track. The reason for this assumption is that a basic computation step for external sorting is merging of two sorted pages (while for an internal sorting algorithm, it is the comparison of two elements). In order for a processor to efficiently perform this operation, its local memory must be large enough to hold at least three pages. If more memory is available, there are two ways to improve the performance of the sorting algorithms. One is to increase the page size (for example, a page size can be a multiple of a track capacity). The other would be to replace the basic 2-page merge by an N-way merge of N pages.

3.3.1. Computation cost

We assume that a full page contains K records. If the length of a page is approximately 47000 bytes [2], the value of K would be about 300 for a record length of 150 bytes. The cost of performing a comparison is denoted by

[2] To obtain numerical estimates for the execution time of the algorithms, we will assume that the disk device characteristics are similar to the IBM 3380 (see section 4.7).

C ; the value of C depends on the length of the sort key. For longer keys, the computation time increases. The cost of moving a record within memory is V time units. This cost depends on the record length. However, because in a 2-page merge all the records are moved (from an input to an output buffer), the number of bytes moved is always $V \cdot K$, that is approximately the number of bytes in a page. Thus, changing the record length does not affect the overall computation time required for moving the records within memory.

Merging two sorted lists of length K requires in the worst-case $2K-1$ comparisons [Knut73]. The number of records to be moved is always $2K$, since all the records must be moved to the output buffer. Thus, the cost of merging two sorted pages is essentially $2K(C+V)$. The computation time of our algorithms will be expressed in T_m time units, where T_m is defined as

$$T_m = K(C+V)$$

When a processor performs a traditional 2-way merge, the computation time required for merging 2 runs of 2^i pages each into one run of 2^{i+1} pages is $2^{i+1}T_m$.

The cost of performing an internal sort of a page is not included in our analysis of the alternative algorithms, since each requires that the pages be internally sorted

before the pages are merged together. We may assume that before each external sorting algorithm starts executing, a preprocessing phase is performed in order to sort individual pages. Or, in order to reduce I/O time, the pages can be sorted when they are read for the first time into the processors' local memory. However, even in the case that an initial phase is performed for the sole purpose of sorting individual pages, the cost of this phase should not affect significantly the overall cost of the algorithms, since at most one pass over the entire file is required to perform it. For a file of n pages, and for p processors, sorting individual pages requires a computation time proportional to

$$(n/p)K \log K (C+V) \quad [3]$$

and a total of $2n$ I/O operations (to read and write the entire file).

3.3.2. Communication time

There are two types of messages passed along the interconnection network. The first type includes page transfers between two processors. Each of these transfers

[3] In all our formulas, the logarithms are base 2, unless otherwise specified. We assume that the pages are internally sorted using a fast internal sort algorithm, that requires $O(K \log K)$ comparisons and record moves.

has a fixed cost, and we have assumed that this cost is equal to the cost of an I/O page transfer (with no access time). The numbers presented in Section 4.7 are based on a 16 ms page transfer cost.

The second type of messages include the control messages: short messages exchanged by the controller and the processors. Examples of control messages are those necessary to allocate processors, synchronization messages indicating the end of a phase, and the initiation of a new phase during execution of an algorithm. Since the number of control messages is small compared to the number of data messages, and since they are short (they contain only a few words of information), we are neglecting them when comparing the cost of several algorithms. The actual cost of control is an open question that we intend to address in the future.

3.3.3. I/O cost

A first indication of the efficiency of a parallel external sorting algorithm is the total number of I/O operations required. An estimation of the time required for I/O also requires, however, consideration of the way a specific algorithm is able to exploit the physical characteristics of the I/O device. In particular, seek time can vary significantly, depending on how intermediate files

created by the algorithm are read and written. For example, seek time can be reduced by filling up one cylinder, and then writing on the next cylinder. If this strategy can be implemented, two entire cylinders can be written with a single track to track seek. An additional improvement can be achieved by performing several track transfers in parallel. Since we are proposing to access, in parallel, all the tracks of the same cylinder, several processors' I/O requests can be simultaneously satisfied provided that the algorithm synchronizes these requests and that concurrent requests are made for data on one cylinder. Ideally, during the entire execution of the algorithm, I/O requests would have this characteristic. As a result, we could estimate the I/O cost by the formula

$$(T_{tr} + T_{sk}) * \text{total transfers} / p$$

where:

T_{tr} is a track transfer time

T_{sk} is the track to track seek time

p is the number of processors allocated for the sort.

It is not the case, however, that all the processors are perfectly synchronized during all the stages of the algorithm. Often, an algorithm cannot use all the processors during certain stages. For example, in the "parallel binary merge" algorithm that will be presented in Section

4.2, one processor must finish its task alone, after the other processors have been released. As a result, during the last stage of this algorithm, I/O transfers must be sequential. Thus, partial processor utilization can substantially limit the amount of I/O parallelism.

An additional constraint that can also result in increasing the seek time and in serializing the I/O transfers, is caused by the "adaptive" nature of our algorithms. By adaptivity, we mean that the sequence of a processor's read requests depends on the data, and is not determined by the algorithm only. For example, for some algorithms, a processor must merge two sorted runs of data, each of which resides on several contiguous tracks of the same disk surface. Depending on the values of the sort keys in both runs, the sequence of read requests can be either:

run 1, run 2, run 1, run 2, ...

or:

run 1, run 1, run 1, run 2, ...

Clearly, the seek time for the first sequence will be higher, since the disk head must skip over the tracks on which run 1 is stored for every other read request.

Because of the complexity of the problem, we decided against having an estimate of the average I/O cost. A better alternative is to consider both a "best case" and a

"worst case" for each algorithm. For the best case, seek time and transfer time are minimized (by assuming that all the active processors read and write tracks in parallel). For the worst case, in all the situations where the algorithm or the data may limit parallelism, we assume a long seek time for each I/O request (typically, half of the tracks on a surface), and serial track transfers. The latter approach will also provide an estimate of the performance of each algorithm when only conventional single channel disks are available.

CHAPTER 4

PARALLEL EXTERNAL SORTING ALGORITHMS

In this chapter, we first describe a modified 2-way merge procedure that has the property of synchronizing its input requests. This procedure will be used as a basic building block for parallel external sorting algorithms that are based on iterated merging. In Sections 4.2 to 4.6, five parallel external sorting algorithms are presented and analyzed: the pipelined binary merge sort and the parallel binary merge sort, which are two parallel versions of the serial 2-way merge sort, the external block bitonic sort, which is based on Batcher's bitonic sort, the pipelined selection sort and finally the broadcast enumeration sort. In Section 4.7, we have grouped numerical results that indicate the execution time of each sort. Based on these results, the algorithms are compared among themselves and with a serial 2-way external merge algorithm. Finally, Section 4.8 is a brief summary of the results presented in this chapter.

Implementation issues and usability of these algorithms were a main concern in our design effort. For this reason, we have previously defined the architecture of the multiprocessors on which the algorithms can be executed

(Section 3.2), and the cost model that is used for their performance evaluation (Section 3.3).

Our algorithms are not based on a well specified, theoretical model of parallel computation. Thus, they are not intended to achieve "optimal speedup" or best asymptotic complexity. Some of the ideas on which these algorithms are based have been suggested before, and they will be recognized by the reader familiar with common approaches to the sorting problem. However, the originality of this research results from addressing for the first time, the problem of external sorting in a parallel environment, and characterizing the architecture and the performance issues that are related to it.

By describing in detail each of our algorithms, we demonstrate how basic building blocks for sorting (such as 2-way or bitonic merge, minimum selection and enumeration) can be used to design parallel sorting algorithms that do not restrict the size of the file to be sorted. However, the parallel external sorting problem raises several issues that are beyond the scope of algorithm design. In particular, the evaluation criteria associated with previous parallel computation models cannot be used to measure the efficiency of parallel external sorting. To motivate the use of parallelism for external sorting, we investigate parallel architectural features and cost parameters that

enable us to perform a fair comparison between parallel external sorting and serial external sorting algorithms.

Unlike this study, most previous research on parallel algorithms and parallel architectures has concentrated on showing how computation time of a complex task can be reduced by the use of parallel processors. As a result little consideration has been given to the I/O problem. Fast parallel algorithms have been developed for solving many problems (including sorting), but when these algorithms were analyzed the cost of reading the data into the multiprocessor's memory (at initiation time of the algorithm) or writing it to mass storage (at termination) was essentially ignored. While neglecting this cost might be justified for tasks that require an extensive amount of computation per unit of data, minimizing I/O time must be a crucial concern when an external sorting algorithm is designed and evaluated. Thus, an important contribution of our study on parallel external sorting is the methodology that we have developed for analyzing the I/O cost of a parallel algorithm.

In order to evaluate the best possible performance of each algorithm, different architecture features have been assumed in each case (within the limits imposed by our model). In particular, since each algorithm dictates specific logical data transfer paths between the

processors, we propose that an interconnection network supply a physical link for materializing these transfers. As a consequence, when the performance of an algorithm is analyzed, a specific interconnection scheme is also evaluated. Therefore, the results of the comparison between the algorithms provides a tool for evaluating architectural features as well as the efficiency of an algorithm.

4.1. Building blocks for parallel external sorting

Three simple approaches may be taken to design an external sorting algorithms: selection, enumeration and bucket sorting. Serial external sorting algorithms belonging to these three categories are known, and they can be extended to generate parallel external sorting algorithms. However, the most common approach to (serial) external sorting is iterated merging. In this section, (after a brief description of the other approaches) we investigate in detail the choice of iterated merging as a building block for parallel external sorting. In particular, we describe and motivate a new synchronous, external merge procedure. We demonstrate that performance of a parallel external sorting algorithm based on iterated merging can be greatly enhanced if the processors execute in parallel this

synchronous procedure.

4.1.1. Selection, enumeration and bucket sorting

A class of simple algorithms can be generated by iterated selection of the minimum element. For an external sort, several passes are required since the entire file does not fit in memory. A straightforward parallelization of this iterated selection process can be achieved by pipelining.

Another class of algorithms is based on enumeration: A count is associated to each record, indicating its rank in the sorted file. In order to establish this count, each record must be compared to all the others. In a parallel environment, many of these comparisons can be performed concurrently by broadcasting a record to several processors. The "pipelined selection sort" and the "broadcast-enumeration sort" that will be presented in the next chapter respectively illustrate these two approaches.

When the range of the sort key values is known a priori, a fast "bucket sort" can be used (whether an internal or an external sort is needed). Here, the idea is to assign a bucket of memory buffers to each of these values. Then, as the source file is read, each record is written into the appropriate bucket. Again, for an external sort, multiple passes are required, because there is not enough

memory to allocate a large enough bucket for each value. A serial multipass sort is described in [Kim80]. Since we are addressing the problem of sorting a large file, without knowing the distribution of the sort key values, we will not investigate parallel versions of bucket sorting.

4.1.2. Iterated merging

Iterated merging is the most common way to perform an external sort on a single processor. As in the serial case, iterated merging can be used to design a parallel external sorting algorithm. However, while serial external sorting algorithms are based on an iterated N-way merge, parallel external sorting algorithms can either use parallel versions of the N-way merge, or a purely parallel merge rule such as the bitonic merge rule (Section 2.2.1). In either case, a basic task that must be performed by each processor is merging two sorted runs of arbitrary length. Therefore, it is important to describe this operation in some detail, and to determine its cost for a given configuration of memory and mass storage device. In this section, we describe how the traditional 2-way merge can be performed by a processor with 3 internal memory buffers. We demonstrate that while input transfers are asynchronous, output transfers are synchronous. While it is not clear whether asynchronous I/O transfers are a disadvantage for a single processor, they may affect the overall performance

when several processors execute an external merge procedure in parallel. For example, if the input runs are stored on a parallel read-out disk (that is a moving-head disk modified to enable parallel reading of the tracks on a cylinder), asynchronous input requests may result in many additional rotations of the disk, that would not be required if the requests were synchronous.

This observation about the traditional 2-way merge has motivated us to develop a new 2-way merge procedure, that has synchronous input requests. This procedure is described below as "the synchronous 2-way merge".

The traditional 2-way merge:

A processor with an internal memory large enough to hold only 3 pages of data, can merge two sorted runs of length n pages (for arbitrary large n) into one sorted run of length $2n$ pages. The input runs are read from a mass storage device, such as magnetic disk or tape, and the output run is written gradually, as it is produced by the processor, to the same type of device. The procedure is known as the "2-way external merge sort" [Knut73]. Initially, the input pages (from both runs) reside on a mass storage device. Upon a request from a processor, one page from either run is transferred from mass storage to the processor's local memory. A one-page input buffer is

assigned to each input run (Figure 15). A one-page output buffer is assigned for writing the output run as it is produced. All three buffers are of the same size, and hold K records. Whenever the output buffer is filled, it is written to mass storage as the next page of the output run. At the termination of the merge procedure, it must also be written to mass storage, even if it is not full. When this merge procedure is executed, the requests for new input pages are issued by the processor in an asynchronous manner. Both the time when an input buffer has been completely scanned, and the identity of the buffer that is exhausted first, are data dependent. For example, suppose that all the values in the first page of the first run are larger than the first value in the first page of the second run. Then, every record from the first input buffer will

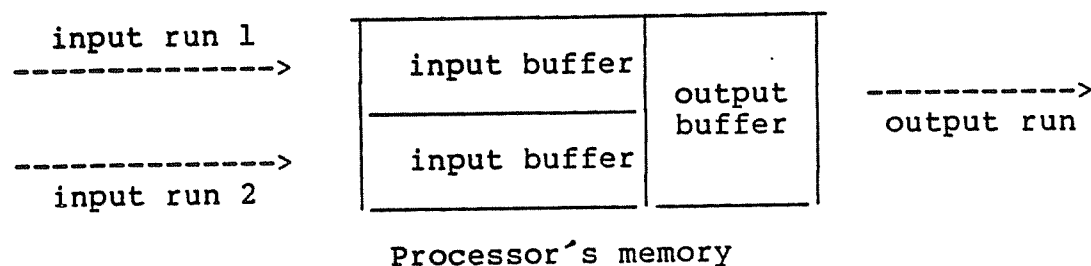
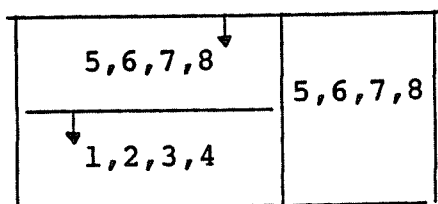


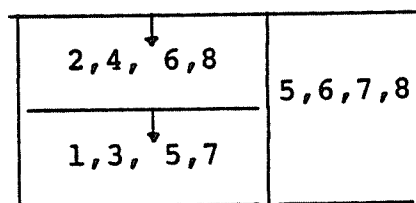
Figure 15. External 2-way merge sort

be transferred to the output buffer before any record from the second input buffer. In this case, an input request for the second page of the first run will be issued after K comparisons and record moves (where K is the number of records per page). On the other hand, if records were moved from both input buffers alternately to the output buffer, no input request would be issued at that time. Both cases are illustrated in Figure 16. On the other hand, the output buffer always fills after the processor has performed K comparisons and K record moves (the moves are from the input buffers to the output buffers). Thus, the output transfers can be considered as synchronous.

A synchronous merge procedure:



Input request issued



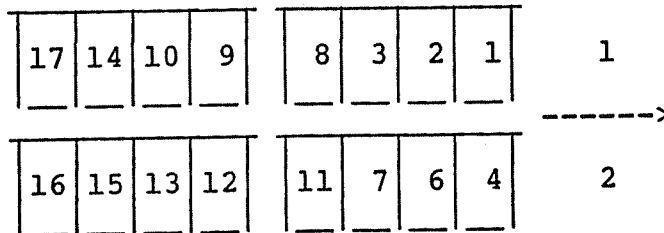
No input request issued

Figure 16. Buffers contents after 4 comparison-move steps

It is possible to synchronize the input requests of a 2-way merge procedure, at the price of more record comparisons. This modified 2-way merge procedure is based on the idea that the pages are always merged by pairs, to produce a sorted run of 2 pages. The first page of this sorted run is written to disk, while the second is kept in memory to be merged with a new input page. In order to determine which of the two input streams is to supply the new page, a "forecasting" mechanism must be implemented. The forecast is made possible by keeping track of the identity of the run from which the next page must be read. To make the following description clear, suppose that this information is kept in register R. When an input page is read, its last element is compared to the last element of the page that previously remained in memory. Depending on the result of this comparison, the contents of register R is either updated or not (see Figure 17). Suppose that R was set to 1. This indicates that the new page comes from the first input run. If the last element of the new page is greater than the last element of the previous page, then R must be set to 2. On the other hand, if it is smaller (or equal), the value in R stays 1, to indicate that the next input page must again be read from the first run.

The time between two consecutive page transfers (input or output) is equal to the time required by the processor

The input runs:



Page in memory

11	8	7	6
----	---	---	---

New page
(from input run 1)

17	14	10	9
----	----	----	---

R=1 17>11, Set R to 2

I

Page in memory

17	14	11	10
----	----	----	----

New page
(from input run 2)

16	15	13	12
----	----	----	----

R=2 16<17, Do not reset R

II

After step II, the new page is read again from input run 2

Figure 17. Forecasting the next input page

to merge 2 sorted pages, that is the time required for $2K$ comparisons and moves, where K is the number of records per page. Some of the elements in the page remaining in memory may be compared to elements of the input run they came from. Thus, these comparisons are redundant. When a traditional 2-way merge is performed, the computation time

required for merging 2 runs of 2^i pages each is $2^{i+1} T_m$. When a synchronous merge is performed, the same operation requires $(2^{i+2}-2) T_m$ (where T_m is the computation time unit defined in Section 3.3). However, the increase in the number of comparisons can be justified when synchronization of the input requests is necessary, or when the processors are fast enough to overlap the computation time with I/O.

4.2. The pipelined merge-sort

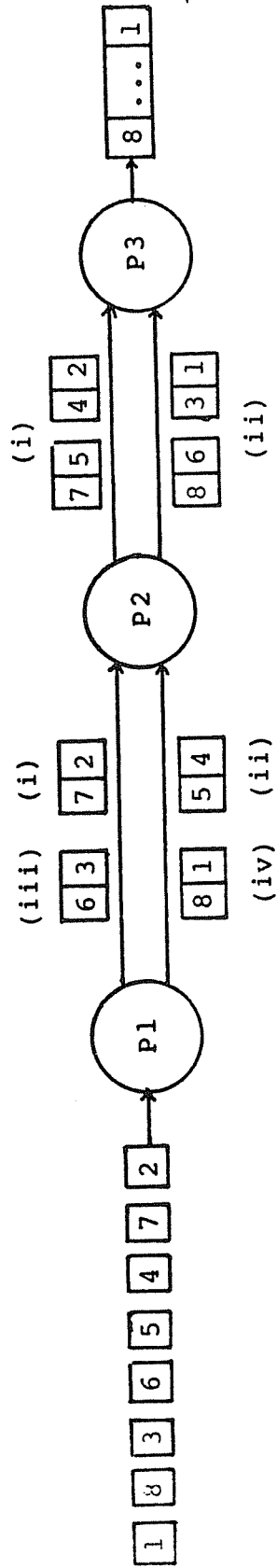
This algorithm is based on Even's parallel tape sorting algorithm (Section 2.4). It is a parallel version of the traditional 2-way external merge, where each phase is supported by a separate processor. An implementation of a similar algorithm for a bubble memory device has also been recently investigated by [Todd78]. We assume that p processors that share access to a common disk device have been allocated to sort a file of n pages. This file is initially stored on consecutive tracks of one disk surface. When the algorithm terminates, the sorted file will be written on consecutive tracks of another disk surface. In order to obtain closed analytical formulas, we assume that n is an exact power of 2. For the general case (when n is an arbitrary integer number), dummy pages can be added to the file and the cost formulas must be accordingly modi-

fied[1].

4.2.1. Description of the algorithm

The processors are labeled P_1, P_2, \dots, P_p , and function as a linear pipeline during execution of the sort. First, Processor P_1 reads pairs of pages from the source file, and merges them into 2-page sorted runs. Then, each processor performs a 2-way merge of pairs of sorted runs produced by its predecessor in the pipeline, as shown in Figure 18. For $1 \leq i < p$, processor P_{i+1} merges pairs of runs of size 2^i pages produced by P_i into runs of size 2^{i+1} pages. Thus, the optimal number of processors required for this algorithm is $p = \log_2 n$, since in this case P_p produces one output run of size n . If less than p are available, additional passes through the pipeline are needed to complete the sort. Each pass, except possibly the last, goes over the entire pipeline, and increases the size of the sorted runs by a factor of 2^p . We consider in detail only the case $n = 2^p$ since even for small values of p , 2^p pages constitute a fairly large file. For example, for $p = 16$ we would be considering files that can be as large as 2^{16} pages, that is about 3,000 Mbyte (for a page capacity of

[1] Since two other algorithms, the parallel binary sort and the block bitonic sort, use the same assumption, a fair comparison can be based on the formulas obtained when n is a power of 2.



a represents a run of 2 pages.
 The first page of the run contains the value 'a' and the second page contains the value 'b'.

Figure 18
 Pipelined Merge Sort
 with
 3 Processors and 8 pages

47,000 bytes). In the case that less than $\log_2 n$ processors are available to sort a file of n pages, our analysis can be extended to estimate the cost of the additional passes. However, this complicates the cost formulas without giving any better insight on the relative performance of the algorithms.

4.2.2. Implementation

The performance of this algorithm depends mainly on how efficiently the transfer of intermediate runs between the processors can be realized. A multiprocessor architecture can be specified, on the basis of the model described in Section 3.2, that allows for an efficient implementation of this pipelined merge (Figure 19). Since in our model the processors' local memory can only hold three pages, each processor must execute an external merge procedure (Section 4.1). Thus, one possibility is to write all the output runs to disk, and to read all the input runs from disk, one page at a time. However, if the processors are directly connected by a linear interconnection network, some pages can be directly transferred along the network links, instead of being first written to disk by a processor and then read by this processor's right neighbor.

Therefore, the logical pipelining process that we have just described can be physically implemented by directly

transferring output runs from processor P_i to processor P_{i+1} . However, since P_{i+1} must merge pairs of runs that are sequentially produced by P_i , every other run produced by P_i must still be written to disk. In order to achieve a maximum amount of overlapping between stages of the pipeline, processor P_{i+1} can begin execution as soon as processor P_i has produced an entire run, and is ready to produce the first page of the second run. At this point, pages from the second run can be pipelined to P_{i+1} . Thus, output runs produced by P_i are alternately written to disk, or directly transferred to P_{i+1} . This implies that P_{i+1} merges one run that is stored on disk with a second run that it receives from P_i . Since during execution of the merge, the next input page requested by P_{i+1} can be either a disk page (from the first run), or a transferred page, P_i may be blocked when it attempts to transfer a page to P_{i+1} . In the extreme case, the data may be such that P_{i+1} receives one page from P_i , but subsequently reads and processes 2^{i-1} pages from the disk, while P_i is waiting to transfer its second page.

Since in our multiprocessor model processors are physically associated to disk surfaces, P_{i+1} can read data written by P_i only if the disk device is designed so that P_i writes on a surface that P_{i+1} can read from. This feature can be achieved by using two disk drives, and by

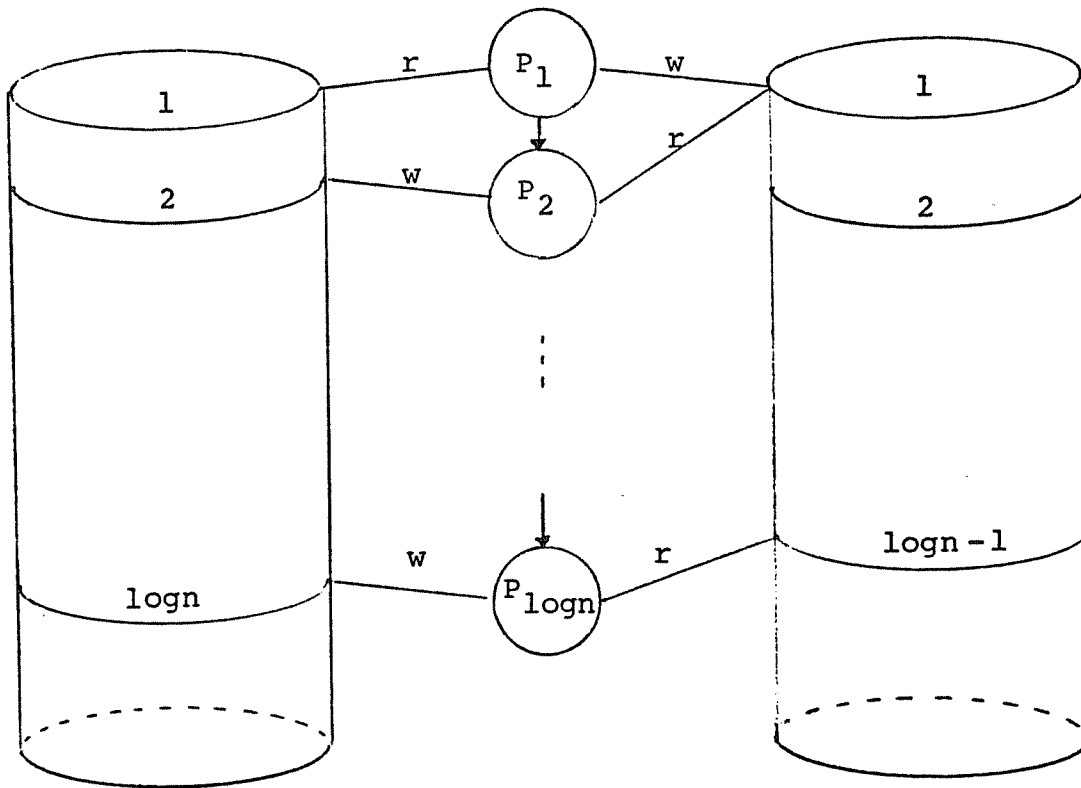
skewing the disk data and control lines with respect to the processors (Figure 19). More specifically, P_1 reads from surface #1 of drive #1, and writes on surface #1 of drive #2. P_2 reads from surface #1 of drive #2, and writes on surface #2 of drive #1, etc. This pattern implies that the source file is initially stored on the first surface of drive #1, and that the result file is written on a single surface of drive #1 or #2 (depending on whether p is odd or even).

4.2.3. Analysis of the algorithm

The computation time can be expressed in T_m units, that is the time required for K comparisons and K record-moves where K is the number of records per page (see Section 3.3). We assume that each processor performs the synchronous merge procedure described in Section 4.1. Thus, processor P_i ($i > 1$) produces an entire output run by merging 2 runs of 2^i pages each in time $(2^{i+2}-2)T_m$, and the first page of another run in time T_m . Since a processor starts execution right after its predecessor has produced an entire run and the first page of a second run, the delay for the last processor P_p is

$$(2^2-1) + (2^3-1) + (2^4-1) + \dots + (2^p-1)$$

Then, P_p merges 2 runs of size $2^{p-1} = n/2$ each in time $(2^{p+1}-2)T_m$. Thus, the total computation time is



$\log_2(n)$ processors
2 disk drives

Figure 19. Architecture for pipelined merge sort

$$\begin{aligned} \sum (2^i - 1) + (2^{p+1} - 2) &= (2^{p+2} - p - 4) \\ &= (4n - \log n - 4) T_m \end{aligned}$$

Communication time:

Every other output run is directly transferred between the processors. Thus, each processor (except the last one) transfers half of the file to its right neighbor. It follows that the total number of page transfers is

$$(n/2)(\log n - 1)$$

A first estimate of the total communication time is given by

$$(n/2)(\log n - 1)T_{tr}$$

where T_{tr} is the time to transfer a page across a network link. However, many of those transfers can be done in parallel. An accurate estimate of the parallel communication time is hard to obtain, because the transfers along several links of the pipeline cannot always be synchronized. However, an optimistic estimate can be obtained by accounting for the time it takes P_1 to transfer half of the file to P_2 (that is $n/2T_{tr}$), plus the time it takes to the last page to reach P_p (that is $(p-2)T_{tr}$), that is:

$$(n/2 + \log n - 2) T_{tr}$$

I/O time:

The entire source file is read from disk by P_1 , and the entire sorted file is written to disk by P_p . If we

assume that the processors are associated to disk surfaces according to the scheme illustrated in Figure 20, then only a run written to disk by P_i can be directly read by P_{i+1} , without requiring additional transfers along the network link. Thus, the total number of page I/O operations is

$$(n+n/2) + (n/2+n/2)(p-2) + (n/2+n) = np+n \\ = n \log n + n$$

In the worst case, all the I/O operations are serialized, and each requires a long seek. Thus, in this case, the total I/O time is:

$$(n \log n + n) * (T_{acc} + T_{tr})$$

I/O time can be substantially reduced if the processors' I/O requests are synchronized so that several tracks on the same cylinder are accessed concurrently. A (somewhat simplified) estimate of the best I/O time can be obtained by estimating the I/O time for the first processor, and adding the number of I/O operations required to propagate the last page through the pipeline. This assumption leads to the following I/O time:

$$[n + 2 \log n - 2] [T_{sk} + T_{tr}]$$

However, it should be noted that this estimate is too optimistic, since the pipeline delays make synchronization of I/O requests very difficult to achieve.

4.3. Parallel binary merge sort

In this section, we describe a parallel 2-way merge sort algorithm which utilizes both parallelism during each phase and pipelining between the phases, to enhance performance. The algorithm requires a binary tree interconnection between the processors. The mass storage device consists of two disk drives, and each leaf processor is associated with a surface on both drives (Figure 20). In addition, the root processor is associated with a surface of one of the two disk drives.

4.3.1. Description and implementation of the algorithm

We assume that there are at least as twice as many pages as leaf processors. That is, there are n pages, q processors, and $n \geq 2(q+1)/2$ [2]. Since a binary tree interconnection is required, q must be equal to $2^i - 1$, for some integer i . Then the number of leaf processors is $p = (q+1)/2$. Execution of this algorithm is divided into three stages as shown in Figure 21. The algorithm begins execution in a suboptimal stage in which sorting is done by successively merging pairs of longer and longer runs until the number of output runs is reduced to p . During this

[2] We make the simplifying assumption that n is a power of 2. If this is not the case, the analysis should be modified by including "dummy" pages, that are processed and transferred in zero time

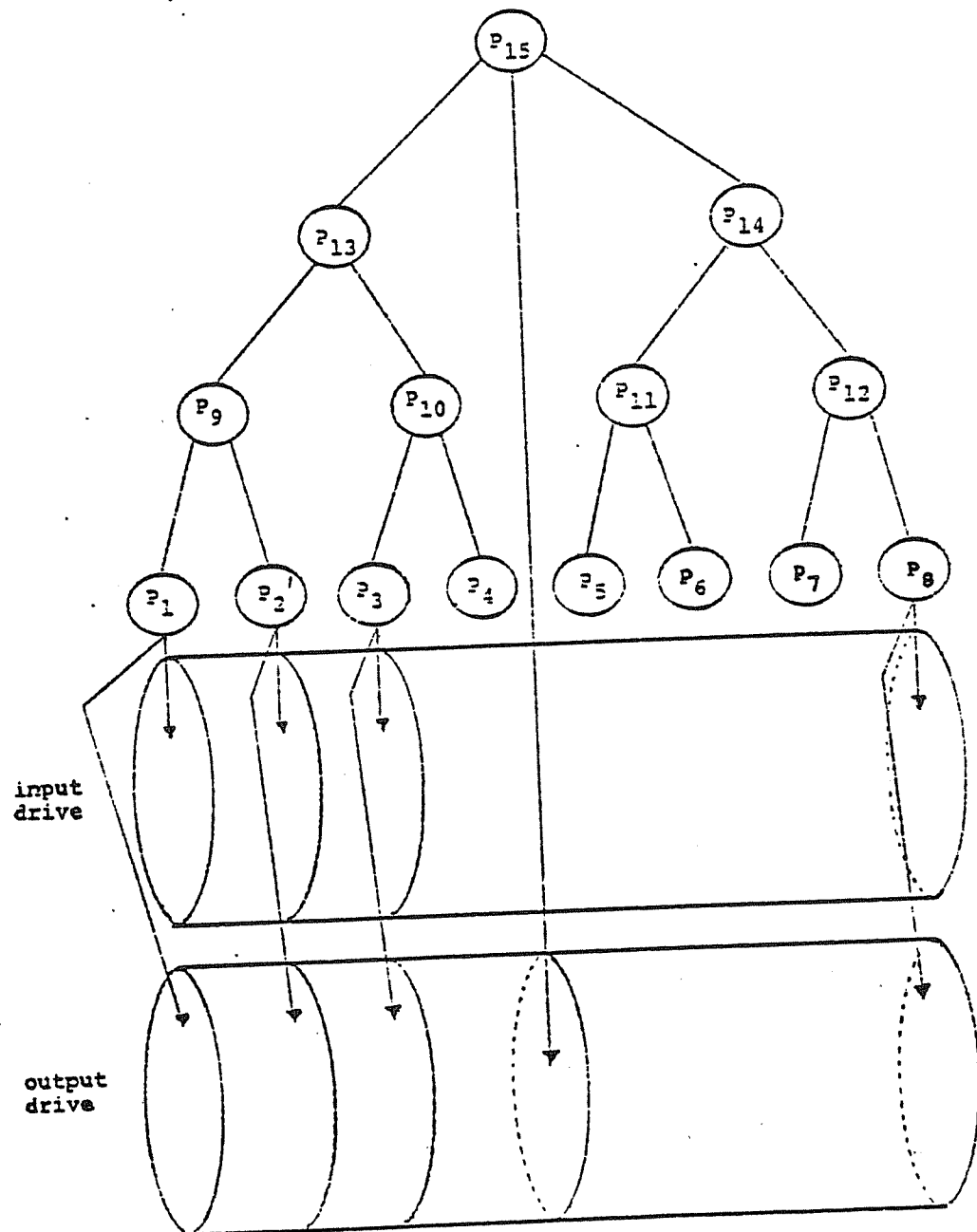


Figure 20. Architecture for the parallel binary merge-sort

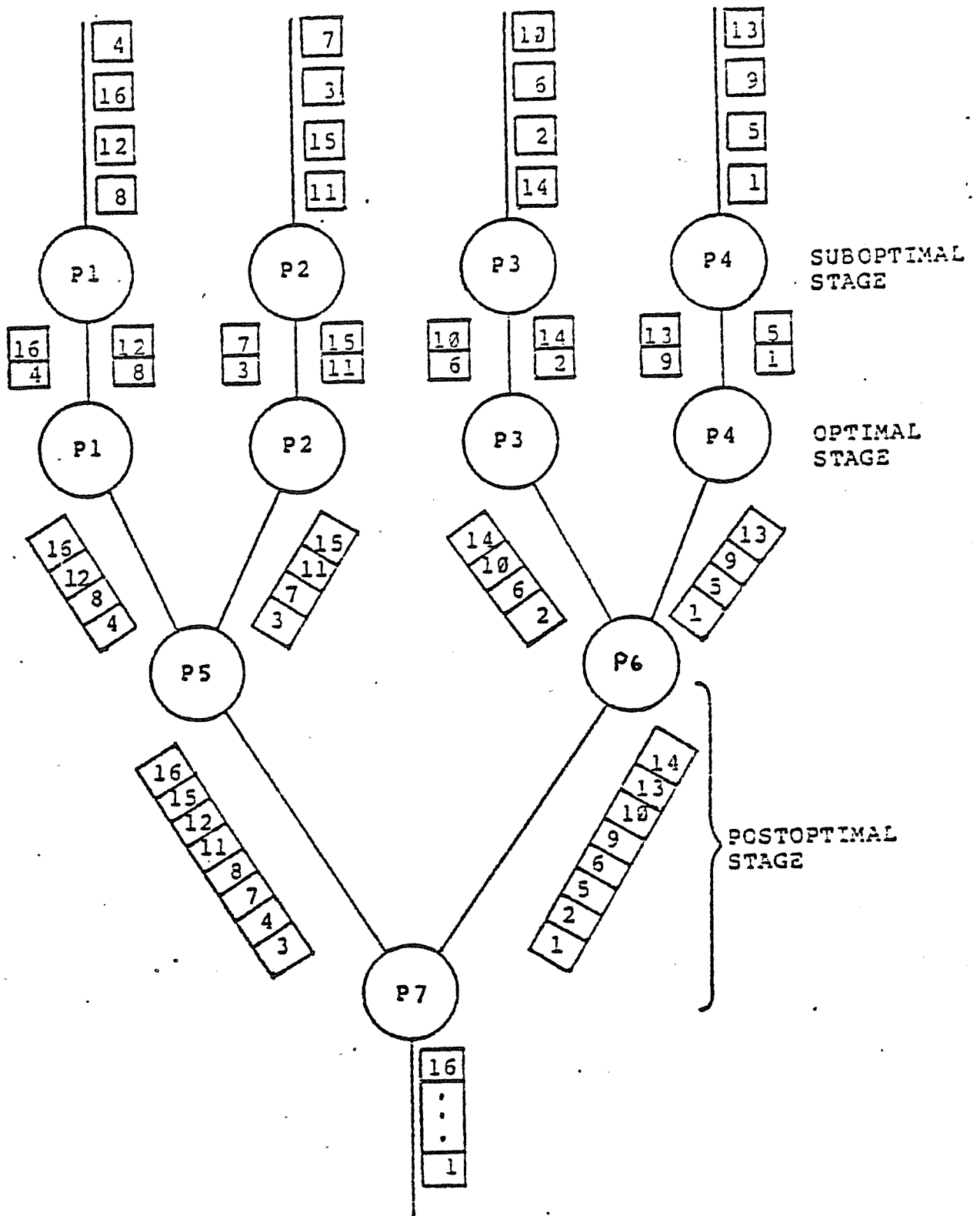


Figure 21. The parallel binary merge-sort algorithm

stage, only the leaf processors execute, while the interior nodes and the root processor are idle. First, each leaf processor reads 2 pages and merges them into a sorted run of 2 pages. This step is repeated until all single pages have been read. If the number of runs of 2 pages is greater than $2p$, each processor proceeds to the second phase of the suboptimal stage in which it repeatedly merges 2 runs of 2 pages into sorted runs of 4 pages, until all runs of 2 pages have been processed. This process continues with longer and longer runs, until the number of runs equals $2p$.

When the number of runs equals $2p$, each leaf processor merges exactly two runs of length $n/2p$. This merge procedure constitutes the optimal stage. When the leaf processors produce the first page (of an output run consisting of (n/p) pages), they transfer it to their parent processor. At that point, the postoptimal stage starts and parallelism is employed in two ways. First, all processors at the same level of the tree concurrently execute a phase of the merge sort (e.g they merge runs of size 2^i into runs of size 2^{i+1}). Second, pipelining is used between levels. All the processors, except the root, directly send each page produced to their parent processor, instead of writing it to mass storage. By pipelining data between levels of the tree, a parent is able to start processing as soon as

its children have produced one page. If a time unit is defined as the time to merge 2 pages and to transfer one page, a parent node receives its first page one time unit later than its children. Therefore, the first two pages will reach the root processor $\log p$ time units after the beginning of the optimal stage. From this point on, the root processors merges the pages it receives and writes the final output run of length n to mass storage.

It is important to note that all the processors cannot operate synchronously. The pipelining process may imply that a child processor is being blocked because its parent is not ready to receive a new page from it. For example, this happens when the data is such that two successive pages of the left input run contain smaller values than the current page in the right input run.

4.3.2. Analysis

During the suboptimal stage, each processor performs $\log(n/2p)$ phases of a serial external merge sort on a file consisting of (n/p) pages. Thus, the time required for this stage is roughly equal to the time it would take a single processor to execute the same merge procedure. There is no communication time, and the computation time is derived as for a serial external merge. Since at each of the $\log(n/2p)$ merge phases of the suboptimal stage a pro-

cessor performs an external synchronous merge (Section 4.1), the parallel computation time for this stage is:

$$\begin{aligned} & (n/p) [2\log(n/2p) - 2 + (4p/n)] \quad [3] \\ & = [(2n/p) \log(n/2p) - (2n/p) + 4] T_m \end{aligned}$$

On the other hand, because the disk device is shared by p processors and the I/O transfers cannot always be parallelized, the suboptimal stage may require more I/O time than a serial merge-sort. Thus, we must consider the two extreme cases: the best case (when I/O transfers are parallel), and the worst case (when each read request results in a long seek and a serial page transfer). The total number of I/O operations is

$$2n(\log(n/2p))$$

The write operations can always be performed in parallel by all the processors, and require only track to track seek time (since a separate disk drive is used for writing). Thus, the time for writing is:

$$(n/p) (T_{sk} + T_{tr}) \log(n/2p)$$

[3] This formula is derived as follows. In Section 4.1, we it was demonstrated that a synchronous merge of 2 runs of size 2^{i-1} requires $(2^{i+1} - 2)T_m$. At the i th merge phase, each processor merges $(n/2^i p)$ pairs of runs of size 2^{i-1} . Thus, the total computation time is obtained by summing the expressions $(n/2^i p) (2^{i+1} - 2)T_m$ for $i=1$ to $\log(n/2p)$.

On the other hand, the time required for reading the input runs at each step can vary, depending on the input runs merge patterns. The best case read time occurs when all the processors can read from the same cylinder, and when the seek time between two input operations is minimal. Thus, the best-case read time is:

$$(n/p) (T_{sk} + T_{tr}) (\log(n/2p))$$

The worst case read time occurs when all the input requests result in a long seek and a single page transfer (that is the page transfers are serialized); thus it is equal to:

$$n(T_{acc} + T_{tr}) (\log(n/2p))$$

During the optimal and the postoptimal stage, each processor (except the root) sends each page it produces to its parent processor. Thus, the interior nodes perform no I/O operations. Pages are read from the disk by the leaf processors and processed at successive levels of the tree. Finally, the root processor writes each output page it produces to disk.

The parallel computation time (for the optimal and the postoptimal stages) is:

$$[2(\log p - 1) + 2n - 2]T_m$$

The communication time can be estimated by summing up the

number of serial page transfers that occur until the first page is ready to be sent to the root processor (that is one transfer per level), and the number of pages transferred to this processor ($n/2$). It is equal to:

$$[(\log p - 1) + (n/2)] T_{tr}$$

The root processor writes the result pages on the disk surface with which it is associated. Since it does not perform any other I/O operations, it can write successive pages with minimal seek time. Therefore, the total I/O time for the optimal and the post optimal stages is equal to:

$$n(T_{sk} + T_{tr})$$

4.4. The external block bitonic sort algorithm

The bitonic sort algorithm uses $n/2$ processors to sort n elements in $1/2 \log n (\log n + 1)$ parallel comparison-exchange steps. This algorithm requires that n (and therefore also the number of processors p) be a power of 2. The processors must be interconnected by a network that can materialize the perfect shuffle permutation and its powers. Some of the networks that have this property have been mentioned in Section 2.2, and we will not investigate their respec-

tive cost and performance. In this section, we describe a parallel external sort algorithm that is based on Batcher's sort. While in Batcher's algorithm, the processing elements were simply comparator-exchange modules (Section 2.2), we now assume that there are p full-scale processors, each having 3 pages of local memory. We also assume that the processors are connected by an interconnection network, such that:

- (1) There is a fixed cost associated with transferring a data element along any of the links required for the bitonic sort (whether the link is materialized by one or several shuffles).
- (2) All the concurrent transfers specified by the bitonic sort can occur in parallel (with no blocking effect).

4.4.1. Description of the algorithm

Batcher's bitonic sort algorithm can be extended to sort a file of n pages, instead of an array of n elements. The basic task performed by a processor is merging 2 sorted pages and exchanging the 2 pages of the sorted block produced (instead of comparing and exchanging 2 elements). The data is transferred by page units, along the links specified at each step by the bitonic sort algorithm. If Batcher's bitonic sort is extended, the resulting algorithm is a "block bitonic sort algorithm", that sorts a file of n pages with $n/2$ processors by executing a sequence of

$1/2 \log_2 n (\log_2 n + 1)$ 2-page operations. Each of these operations consist of merging 2 sorted pages and transferring the result pages to 2 destination processors. Thus, the processors must have 4 pages of local memory to merge 2 pages within this memory. The procedure is illustrated for $n=4$ in Figure 22. A further extension of the block bitonic sort can generate a parallel external sort algorithm. Suppose that the file to be sorted has n pages and that p processors are available where $p < n/2$. As with the parallel binary merge sort, $2p$ sorted runs of $n/2p$ pages each can be produced by a preprocessing stage. During this stage, each processor operates independently on an equal portion of the source file, and produces 2 sorted runs of size $n/2p$. Then, an external block bitonic sort can merge these runs and produce the result file. The number of steps required is still equal to $1/2 \log_2 2p (\log_2 2p + 1)$. Only now, the basic task performed by a processor must be an external merge of 2 runs of size $n/2p$. This output run is then split in half, and each of the two halves becomes an input run for a different processor at the next step. It is important to synchronize the processors at the end of each step so that complete runs are produced before the next step starts. Figure 22 illustrates how this generalized bitonic sort produces a sorted file.

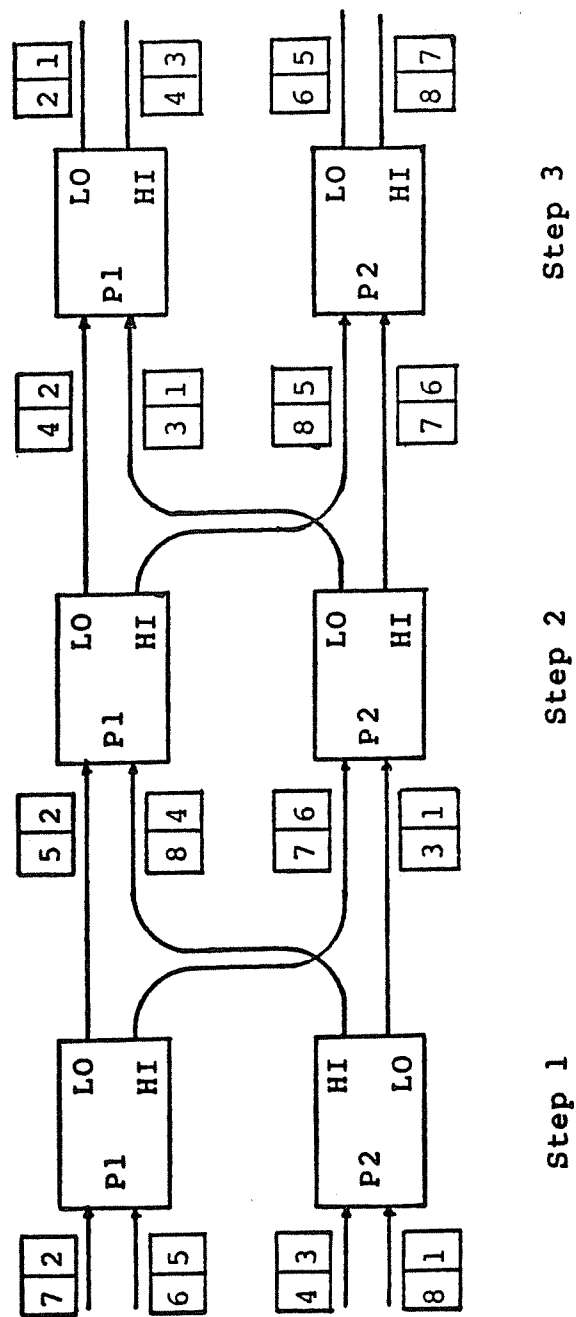


Figure 22. The Block Bitonic Sort Algorithm

4.4.2. Implementation

The main problem in implementing this external block bitonic sort is that a processor must write each half of its output run so that it can be read by another processor at a subsequent step. For example, in the case $p=8$ and $n=32$, during the first step processor P_2 produces an output run of 4 pages; during the second step, the first 2 pages of this run must be read by P_1 , and the last 2 pages must be read by P_3 . According to our multiprocessor model, the processors are interconnected by an interconnection network, and each processor is physically associated with a surface of two disk drives. If this network is the shuffle-exchange network (or any other network that can materialize the bitonic sort transfers), then the output runs can be transferred, one page at a time, as they are produced to the destination processors. The configuration of a processor's local memory is as shown in Figure 23. During a step, data is read from one disk drive into the input buffers and written from the transfer buffer to the other disk drive (on the surface associated with the processor). When the output buffer fills, its contents is transferred to the transfer buffer of a destination processor. At the next step, the write drive becomes the read drive, and conversely. Because of the way the output buffers contents were transferred, each processor can now

read its input runs from the surface it is associated with.

Although the addition of a transfer buffer solves the problem of transferring the runs between disk surfaces, it would be more efficient to implement the algorithm on a different architecture (Figure 13). Because the output runs produced at the i th step must be written to disk, and then read from disk at the $(i+1)$ st step, what is really required is that a processor be able to read and write on any disk surface. Thus, rather than providing an intercon-

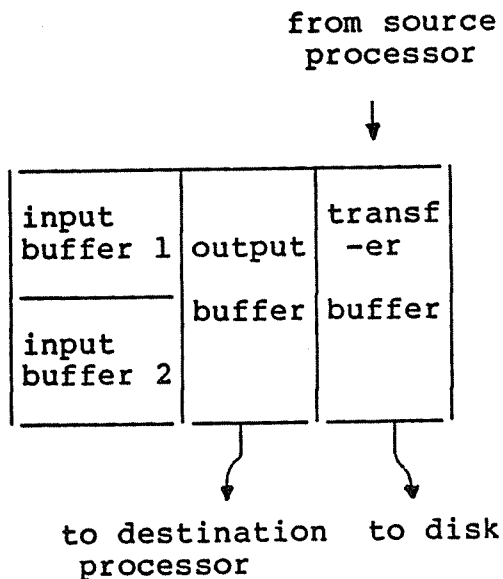


Figure 23. Processor's memory for external block bitonic

nection network between the processors, the interconnection network should connect the disk surfaces to the processors. In this case, each step of the external bitonic sort can be implemented with exactly n/p parallel page write operations, since the output runs can be written directly to those surfaces from which they will be read during the subsequent step.

4.4.3. Analysis

The analysis of the preprocessing stage (that produces $n/2p$ sorted runs) was previously presented for the parallel binary merge algorithm (Section 4.3). However, for the sake of completeness, we briefly repeat the results that were obtained.

$$T(\text{computation}) = [(2n/p) \log(n/2p) - (2n/p) + 4] T_m$$

$$T(\text{communication}) = 0$$

In the best case:

$$T(I/O) = [(2n/p) \log(n/2p)] (T_{sk} + T_{tr})$$

In the worst case:

$$T(I/O) = [(n/p) (T_{sk} + T_{tr}) + n(T_{acc} + T_{tr})] \log(n/2p)$$

After the suboptimal stage, the bitonic sort stage is performed. There are $1/2 \log_2 p (\log_2 p + 1)$ identical steps, during which each processor performs an external synchronous merge on 2 runs of $n/2p$ pages. Therefore, for this stage

$$T(\text{computation}) = ((2n/p) - 2) T_m [1/2 \log_2 p (\log_2 p + 1)]$$

Depending on which of the two types of architecture (described in the implementation section) is used, the communication time is either null (if the processors can access any disk surface) or is equal to

$$T(\text{communication}) = (n/p) T_{tr} [1/2 \log_2 p (\log_2 p + 1)]$$

In order to estimate the I/O time, we observe again that writing the output runs requires only track to track seek time for each page, while the time for reading the input runs can vary between a lower bound (the best case) and an upper bound (the worst case). The time for writing the output runs is:

$$(n/p) [T_{sk} + T_{tr}] [1/2 \log_2 p (\log_2 p + 1)]$$

To estimate the best and worst-case read time, we observe that at the beginning of a new step, all the p input runs are aligned. Thus, the first p pages can be transferred in parallel. However, after the first pages of all the input runs have been read, subsequent read requests may require a long seek and a serial transfer. Thus, the best and worst-case read time are respectively equal to:

$$(n/p) [T_{sk} + T_{tr}] [1/2 \log_2 p (\log_2 p + 1)]$$

and

$$(2+n-2p) [T_{acc}+T_{tr}] [1/2 \log_2 p (\log_2 p + 1)]$$

Some account must also be made for the cost of synchronizing the processors at the end of each step (since a step cannot be initiated unless all the output runs of the previous run have been completely written).

4.5. The pipelined selection sort

Unlike all the previous algorithms, this algorithm does not produce longer and longer runs during intermediate steps of its execution. Thus, it does not require that the processors merge blocks longer than a single page. Basically, the algorithm is based on iterative selection. The "minimum" of n pages is determined, then the "minimum" of the remaining $n-1$ pages is determined, and the operation is repeated until the "maximum" page is created. By minimum page, we mean the page that contains the K lowest records. Thus, the order defined on pages is a partial order. To create a total order on a set of individually sorted pages, the pages must be merged. Merging a pair of pages produces a sorted block of 2 pages; the first page of this block can be designated as the "smaller" page. If it is subsequently merged with a new page, the first page produced will now contain the lowest K records among the $3K$ source records.

4.5.1. Description of the algorithm

Parallelism is introduced by having one processor assigned to each step of minimum selection. In other words, the first processor selects the minimal page among n pages, the second processor selects the minimal page among the remaining $n-1$ pages, etc. If enough processors are available, the algorithm performs optimally when $p=n$ processors are assigned to the sort operation. In this case, the processors are labeled P_1, P_2, \dots, P_n and logically organized as a pipeline.

P_1 reads the first page of the source relation, then it repeats $(n-1)$ times the following procedure: It reads a new page, merges it with the page that was previously in its local memory, and sends the greater page (that is the second page of the 2-page sorted block) to P_2 .

P_2 starts execution after it has received 2 pages from P_1 . It executes the same sequence of steps as P_1 , except that it receives pages sent by P_1 instead of reading from disk. After P_2 has received 2 pages, it starts processing in the same way and sends its lower page to P_3 .

As the pipeline is filled, the pages flow one at a time through the processors. When the last page reaches P_{n-1} , P_i , for $i=1, 2, \dots, n-2$, contain the i th page of the sorted relation. The file is completely sorted when a page

reaches P_n . This page is the maximum page, and at that point, for $1 \leq i \leq n$ P_i contains the i th page of the sorted file. Thus, the result file can be written in parallel by all the processors on a single cylinder.

In the general case, when $p < n$, the algorithm requires multiple phases. Each phase repeats the basic linear pipeline algorithm, except that the last processor must write to disk excess pages that no other processor can receive. During the first phase, this creates a "bucket file" of $(n-p)$ pages which is not sorted. On the other hand, the p pages residing in P_1, P_2, \dots, P_p constitute the first p pages of the sorted relation. To enhance performance, the direction of the pipeline should be reversed between successive phases, so that the processor that wrote the bucket pages can read them at the subsequent phase. For $n=kp$, the algorithm will require k phases with each phase producing p pages of the sorted relation. If n is not an exact multiple of p , then the only modification is for the last phase: If less than p pages are left then the last phase uses a shorter pipeline of length $(n \bmod p)$.

4.5.2. Implementation

Only the first and the last processor read from disk (during the first phase, the source file and during each subsequent phase, the bucket file). However, if each pro-

cessor is associated with a disk surface, all the processors can simultaneously write their result page at the termination of each phase. Transfers between processors can be efficiently implemented by a ring-like architecture that allows for simultaneous transfers from a processor to its right neighbor. The architectural features required for this implementation are illustrated in Figure 24. The processors' local memory must contain at least 4 page buffers (Figure 25), since a processor must merge a pair of pages and keep the first page produced by this merge operation. At any step, two buffers contain the pages being merged, and either the first result page (that must remain in memory) or the second result page (that must be transferred to the next processor) are being filled. During the next step, that page that remained in a destination buffer becomes one source page. The second source page is the page that is received from the left-hand neighbor.

4.5.3. Analysis

At any phase of the algorithm, the first processor on the pipeline (P_1 or P_p) performs the longest computation, since it merges pairwise all the pages of the bucket file. Thus, the parallel computation time can be obtained by considering only the computation time of this processor. At Phase i , the bucket file has $(n-ip)$ pages. Thus it requires time $2(n-ip-1)T_m$ to compute the minimal page. The

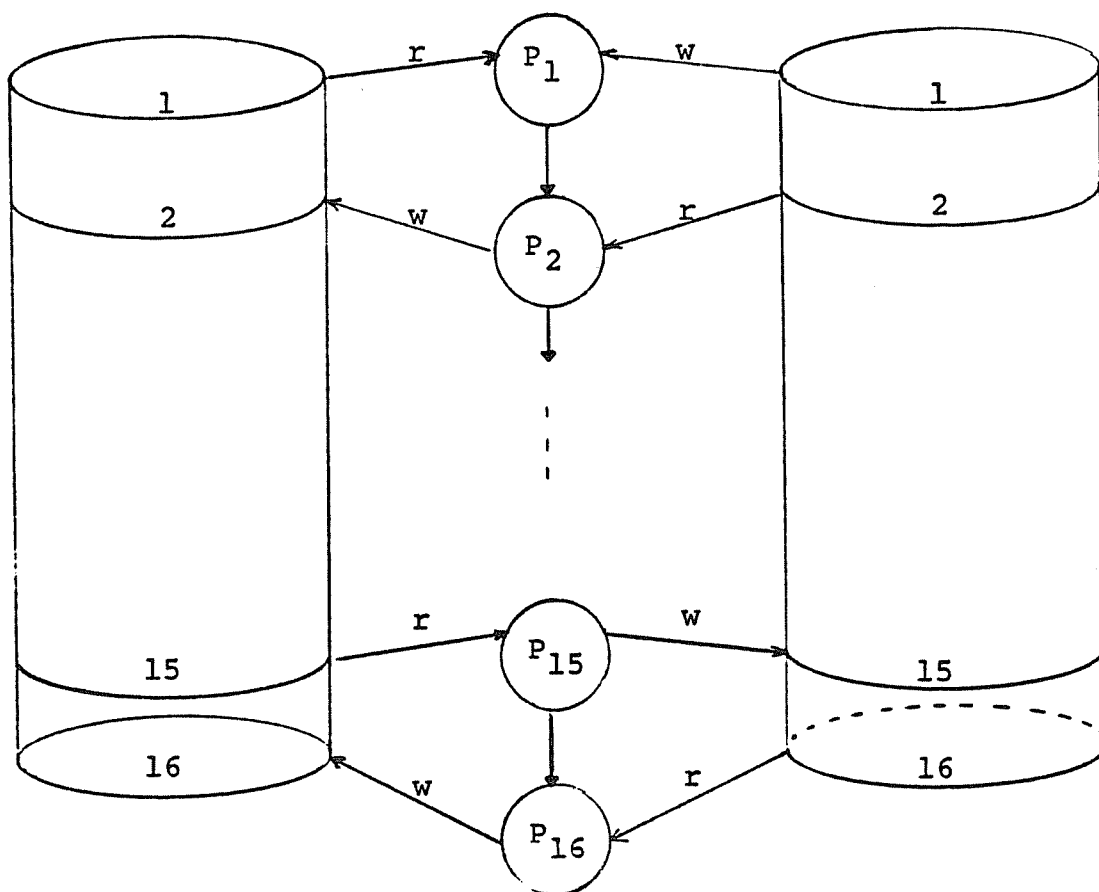


Figure 24. Architecture for pipelined selection sort

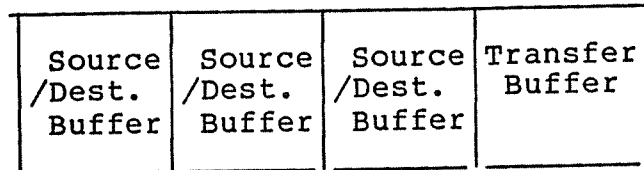


Figure 25. Processor's memory for pipelined selection sort

computation time for all the phases in T_m units is:

$$i=k-1$$

$$\sum_{i=0}^{k-1} 2(n-ip-1) = 2(pk(k+1)/2 - k) = n^2/p + 2n - 2n/p$$

$$i=0$$

The communication time can be estimated in a similar manner, since all the processors (except the last on the pipeline) transfer a result page after every 2-page merge. Thus

$$T(\text{communication}) = [n^2/2p + 3n/2 - 3n/p] T_{tr}$$

Finally, the number of pages read and written at the i th phase is $(n-ip)$, for $i=0,1,\dots,k$. Thus the total number of I/O operations is

$$n^2/p - n/2$$

However, since at the end of a phase p pages are written in parallel, only the rest of the I/O transfers must be performed serially by P_1 and P_p . Thus the I/O time required by the i th phase is:

$$\begin{aligned}
 &T_{\text{acc}}+T_{\text{tr}} && \text{(read the first page)} \\
 &+(n-ip-1)(T_{\text{sk}}+T_{\text{tr}}) && \text{(read the other pages)} \\
 &+(n-ip)(T_{\text{sk}}+T_{\text{tr}}) && \text{(write bucket file)} \\
 &+p(T_{\text{sk}}+T_{\text{tr}}) && \text{(write result pages)}
 \end{aligned}$$

The total I/O time is obtained by summing these expressions for $i=0$ to $(k-1)$:

$$T(\text{I/O}) = (n^2/p)(T_{\text{sk}}+T_{\text{tr}}) + (n/p)(T_{\text{acc}}-T_{\text{sk}})$$

Note that this algorithm does not require an estimate of best and worst case I/O time, since most of the I/O transfers must be serialized (for reading and writing at the end of the pipeline), but the others can always be performed in parallel (for writing the result pages, p at a time). However, if n is significantly larger than p , the number of page I/O operations this algorithm performs is $O(n^2/p)$. Thus it is much slower than the algorithms presented in the previous sections (Section 4.1 to 4.3), that required only $O(n \log n/p)$ I/O operations.

On the other hand, this algorithm has several advantages. In particular, it appears to be simple to imple-

ment, and requires no storage overhead. Unlike the previous sorting algorithms, it does not require that the controller maintain page tables for temporary files or complex control tables for processor reassignment. Another major advantage is that the first pages of the sorted pages are produced very fast (after a single pass over the source file). Thus, this algorithm may be a good choice when sorting is only an intermediate operation. For example, if sorting is performed in a relational database management system, it may be the case that the sorted relation must be joined with another relation. In this situation, performance can be enhanced if this sorting algorithm is used and the result pages are joined, as they are produced, with pages of the second operand relation.

4.6. The broadcast-enumeration sort

Sorting is performed by enumerating for each record with key value v the number of records with a key value smaller than v . Once this enumeration has been performed for all records, the file can be sorted by gathering the records in their count order. If there are enough processors available (that is if $p=n$), the records may be routed so that the i th page of the sorted file will reside in the i th processor's local memory. Parallelism is used in both

phases : the enumeration phase and the routing phase.

4.6.1. Description and implementation

Enumeration phase: Initially each processor reads a page of the source relation. As previously stated, we assume that the pages are internally sorted. An additional integer field (the "count" field) is appended to each record, and is initialized to the value zero. Next, the file is broadcast, one page at a time to all processors. As a broadcast page is received, every processor updates its count for each of the records in its page by comparing the records in the broadcast page with the records in its own page. Since the pages are individually sorted and there are K records in a page, the count can be computed with no more than $2K$ record comparisons (this operation has the same complexity as the merge of 2 sorted lists of length K). The procedure used to compute the count field must guarantee that each record receive a unique count value, despite the fact that there may be duplicate keys in the source relation. A way to achieve this is to increment the count of a record found equal to a broadcast record only if it belongs to a page numbered higher than the broadcast page or if it belongs to the same page but comes after the broadcast record in this page.

The following example illustrates this procedure. Suppose that the keys are initially distributed in the following manner:

P1	P2	P3	P4																
<table border="1" style="border-collapse: collapse; width: 40px; height: 40px;"> <tr><td style="padding: 5px;">2</td><td style="padding: 5px;">2</td></tr> <tr><td style="padding: 5px;">_</td><td style="padding: 5px;">_</td></tr> </table>	2	2	_	_	<table border="1" style="border-collapse: collapse; width: 40px; height: 40px;"> <tr><td style="padding: 5px;">1</td><td style="padding: 5px;">1</td></tr> <tr><td style="padding: 5px;">_</td><td style="padding: 5px;">_</td></tr> </table>	1	1	_	_	<table border="1" style="border-collapse: collapse; width: 40px; height: 40px;"> <tr><td style="padding: 5px;">1</td><td style="padding: 5px;">1</td></tr> <tr><td style="padding: 5px;">_</td><td style="padding: 5px;">_</td></tr> </table>	1	1	_	_	<table border="1" style="border-collapse: collapse; width: 40px; height: 40px;"> <tr><td style="padding: 5px;">4</td><td style="padding: 5px;">4</td></tr> <tr><td style="padding: 5px;">_</td><td style="padding: 5px;">_</td></tr> </table>	4	4	_	_
2	2																		
_	_																		
1	1																		
_	_																		
1	1																		
_	_																		
4	4																		
_	_																		

The counts will then be equal to :

4 5	0 1	2 3	6 7
-----	-----	-----	-----

When the enumeration phase terminates, a count field has been appended to all records. This field indicates their position in the sorted file. When there are less processors than pages, the enumeration phase requires $\lceil n/p \rceil$ steps. During each step, the processors compute the count for the records in p pages and write out the modified p pages (where the count has been appended to each record).

Routing phase: During this phase of the algorithm, the K smallest key records are routed to P_1 , the next K to P_2 , etc. If $p=n$, the routing phase terminates with the records distributed, in order, among the processors. Otherwise routing requires $\lceil n/p \rceil$ steps, where each step produces p pages of the sorted relation. Routing also requires a broadcast of the entire relation. The processors broadcast their page, in sequence. After a page has been broadcast, all processors scan it and copy to an internal buffer records that "belong" to them. That is, processor j copies

a record from the broadcast page to its output buffer if the record count satisfies the inequalities:

$$(j-1)*m*K < \text{count} \leq j*m*K$$

where m is the step number and K is the number of tuples per page. Thus, each page broadcast is followed by the following operations executed in parallel by all processors: a scan of the broadcast page (to select the appropriate records) and an internal move of the selected records. After the entire file has been broadcast, the processors write their output buffer and initiate the next step.

The architecture proposed for this algorithm is shown in Figure 26. It is assumed that at the time the algorithm is initiated, the entire file resides on the disk surface associated to the first processor. During intermediate stages of the algorithm all the processors concurrently write their result page on a cylinder of the output drive.

4.6.2. Analysis

For the enumeration phase, each page broadcast is followed by approximately $2K$ key comparisons and $2K$ increments of the count fields. If we group these operations and designate them as T_m' , we conclude that during the enumeration phase the computation time can be expressed as:

$$T(\text{computation}) = [(n/p)n]T_m'$$

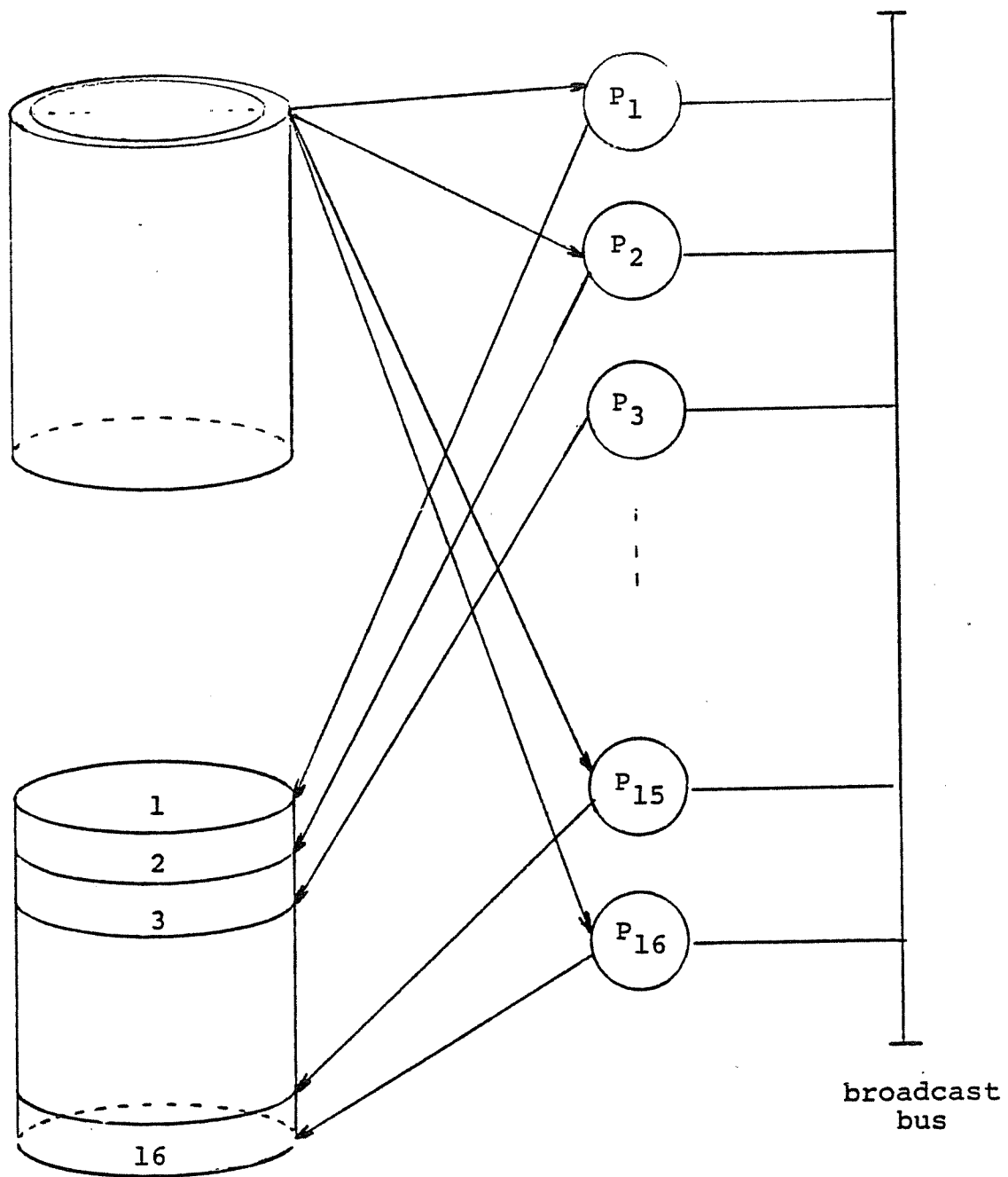


Figure 26. Architecture for broadcast-enumeration sort

In the routing phase, a page broadcast is followed by at most K integer comparisons (to check if the count falls in the range assigned to the processor) and by K/n record moves, on the average. In addition, after all records have reached their destination processor, each processor must perform an internal sort of the records residing in its buffer. Note, however, that the internal sort may use the counts that were previously computed, so that the cost of sorting will be K record moves (KV) rather than $K \cdot \log K$ record comparisons and moves. Thus, during the routing phase

$$\begin{aligned} T(\text{computation}) &= (n/p) [n(KC + (K/n)V) + KV] \\ &= (n^2/p) [KC] + (2n/p) [KV] \end{aligned}$$

The communication cost is the cost of broadcasting the entire file n/p times for each phase (once for the enumeration phase, the other for the routing phase). Thus,

$$T(\text{communication}) = [2(n/p)n]T_{tr}$$

In the optimal case ($p=n$), n pages are read and written once, in parallel, for each of the two phases. In the general case ($p < n$), the entire file must be read at each of the (n/p) steps of each phase. During a step of the enumeration phase, p pages are read in parallel by the processors, then the remaining $(n-p)$ pages are read and broadcast sequentially. Finally, p modified pages (with a count

appended to each record) are written in parallel. During a step of the routing phase, n pages are read serially, then p pages (of the sorted file) can be written in parallel. If two disk drives are used (one for reading and the other for writing), the parallel write operations can be performed with a minimal seek time (i.e. only a track to track seek). For the enumeration phase, the parallel read operation at the beginning of a step requires a long seek, but the following serial read operations (except the first) require only a track to track seek. We conclude that the total I/O time is:

$$\begin{aligned}
 & (n/p) [2T_{acc} + 2T_{tr} + (n-p-1)(T_{sk} + T_{tr}) + T_{sk} + T_{tr}] \\
 & \quad \text{(enumeration)} \\
 & + (n/p) [n(T_{sk} + T_{tr}) + (T_{sk} + T_{tr})] \\
 & \quad \text{(routing)} \\
 & = (2n^2/p - n + n/p)(T_{sk} + T_{tr}) + 2(n/p)(T_{acc} + T_{tr})
 \end{aligned}$$

Like the pipelined selection sort, the broadcast-enumeration sort requires $O(n^2/p)$ I/O operations (when p is significantly less than n). However, the access time is relatively low, because the processors always request consecutive pages during execution of the algorithm.

4.7. Performance comparison of the 5 sorting algorithms

Based on the analytical formulas that were developed in Sections 4.2 to 4.6, we have estimated the execution time of each algorithm, under various assumptions about the processors' capabilities, the disk device characteristics and the file size. Because of their complexity, the cost formulas alone do not provide a clear understanding of the algorithms performance. For this reason, we felt that it was necessary to determine the parallel sorting times for a large file (100 thousand records of 100 characters each, for example), and to compare them to the serial sorting time for the same file. According to statistics that have been collected a few years ago on high-performance computer systems, tape-sorting of a file of this size takes from 15 to 19 minutes [KNUT73, p338], and external disk-sorting takes about 12 minutes [KNUT73, p.309]. How much faster can each of our parallel external sorting algorithms perform the same task? Rather than attempting to perform an asymptotic complexity comparison of our algorithms (among themselves and with a serial sorting algorithm), we have chosen to present a numerical comparison. In this section, we first describe how a value (or a range of values) was assigned to the parameters that define our analytical model. We then present estimates for the computation, the communication and the I/O time of each algorithm, and com-

pare the performance of the five algorithms. We also compare the algorithms to a serial, external 2-way merge sort and conclude this chapter by giving an indication of the parallel speedup that the best algorithm provides.

4.7.1. Description of the parameters values

The cost formulas that have been developed for each of the algorithms in this chapter are expressed in terms of two categories of parameters. Parameters in the first category characterize the I/O device and the multiprocessor. They have been defined as part of our cost model, in Section 3.2. The second category characterizes the file size and to its structure. In this section, we describe and justify the values that we have chosen for both categories of parameters.

Disk device characteristics:

The analysis of our algorithms was strongly impacted by modelling the I/O device as a modified moving-head disk (that allows for parallel read/write of tracks on the same cylinder). However, we felt that our assumptions about track capacity and track transfer rate should be based on the values provided by fast, but conventional moving-head disks. If parallel read/write disks become available, it is reasonable to assume that they will provide similar values for these two parameters. We have based our

estimates of I/O time on the specifications of the IBM 3380 moving-head disk device. Originally, we had also considered the IBM 3350 disk, but the execution time of the algorithms was significantly higher than what will be presented below. The IBM 3380 is characterized by its very large track capacity (more than twice the capacity of most other disks) and a low average seek time. Table 4.6.1 summarizes the specifications of this device.

Table 4.6.1: IBM 3380 Disk Specifications

parameter	symbol	value
Track capacity	-	47,476 bytes
Rotation time	T_{tr}	16.7ms
Tracks per cylinder	-	15
Transfer rate	-	3Mbyte/sec
Average seek time	-	16ms
Track-to-track seek time	T_{sk}	5ms [4]

Detailed estimates of the algorithms execution time have been computed for the case $p=32$ (the number of processors).

[4] Except for the value of T_{sk} , all the numbers in this table have been taken from the IBM 3380 Direct Access Manual. However, since the track to track seek time was not explicitly specified, we had to extrapolate from the other specifications to obtain this 5ms estimate.

Thus, we assume that at least 32 tracks can be read or written in parallel.

Processors' capabilities:

The values that were assumed for the basic parameters were:

- C - the time to compare 2 keys: 10 to 100 microseconds (depending on the key length).
- V - the time to move a record, within a processor's memory: is based on the cost of 1.5 microseconds to move a single word. Thus, for a record length of 150 bytes, V is 225 microseconds.
- T_m - the merge cost per page; since it is equal to $(C+V)$ times the number of records per page, the values that we are considering for this parameter are in the range of 35 to 70 ms.

File size and record length:

The size file was varied from 32 to 16384 (2^{14}) pages. With a page capacity of 47,476 bytes (the IBM 3380 track capacity), this range corresponds to a range of file sizes from 7000 to 4 million 200-byte records. The record lengths considered were between 100 and 400 bytes. The key length was varied from 20 bytes to the entire record length. Varying these two parameters enabled us to observe the impact of higher computation time per page. Our conjecture was that parallel external sorting would be more efficient if the ratio between computation time and I/O time was increased. For a fixed number of processors,

while the amount of parallelism that can be used during computation steps is limited only by the algorithm, for the I/O transfers it is limited both by the algorithm and by the I/O device characteristics. Thus, as expected, the enhancement in performance (shown by the parallel algorithms, compared to a serial external merge) was more significant when the computation time per page was higher. At the extreme, when the computation time per page was low (less than 40ms), the worst case performance of the block bitonic sort became only marginally better than the serial sort performance.

4.7.2. Results of the experiments

Computation time:

If performance was not limited by the physical characteristics of the I/O device, the total execution time of a parallel external sorting algorithm would be roughly proportional to the number of 2-page merge operations required (since every page is read before it is merged with another page and a result page is written after the 2-page merge). Thus, in this ideal situation, we could compare the algorithms by estimating their computation time only. We have plotted this time in T_m units as a function of the file size, when the number of processors is equal to 32 (Figure

27). Six curves are shown in this graph: one for each of the parallel algorithms, and one representing the serial 2-way merge. The file size was varied from 32 to 16384 pages (that is from 1.5Mbyte to 800Mbyte). The lowest computation time is obtained for the block bitonic sort. Next come the parallel binary merge sort, then the pipelined merge sort. The curves representing the computation time of these three algorithms get closer to each other as the file size becomes larger. For medium size files (64 to 512 pages), the pipelined selection sort and the broadcast-enumeration sort perform relatively well (compared to the serial sort), but for very large files, they require even more computation time than the serial sort. The reason for this very poor performance for large values of n , is that the serial sort is a fast algorithm (of order $O(n \log n)$), while the complexity of the pipelined selection sort and the broadcast-enumeration sort is $O(n^2/p)$.

As an indication of the parallel speedup that can be achieved (for computation time only), let us consider the case $n=1024$ (that is, 32 times more pages than processors). In this case, the block bitonic sort and the parallel binary sort require respectively 13 and 8 times less computation time than the serial sort.

Total number of page operations:

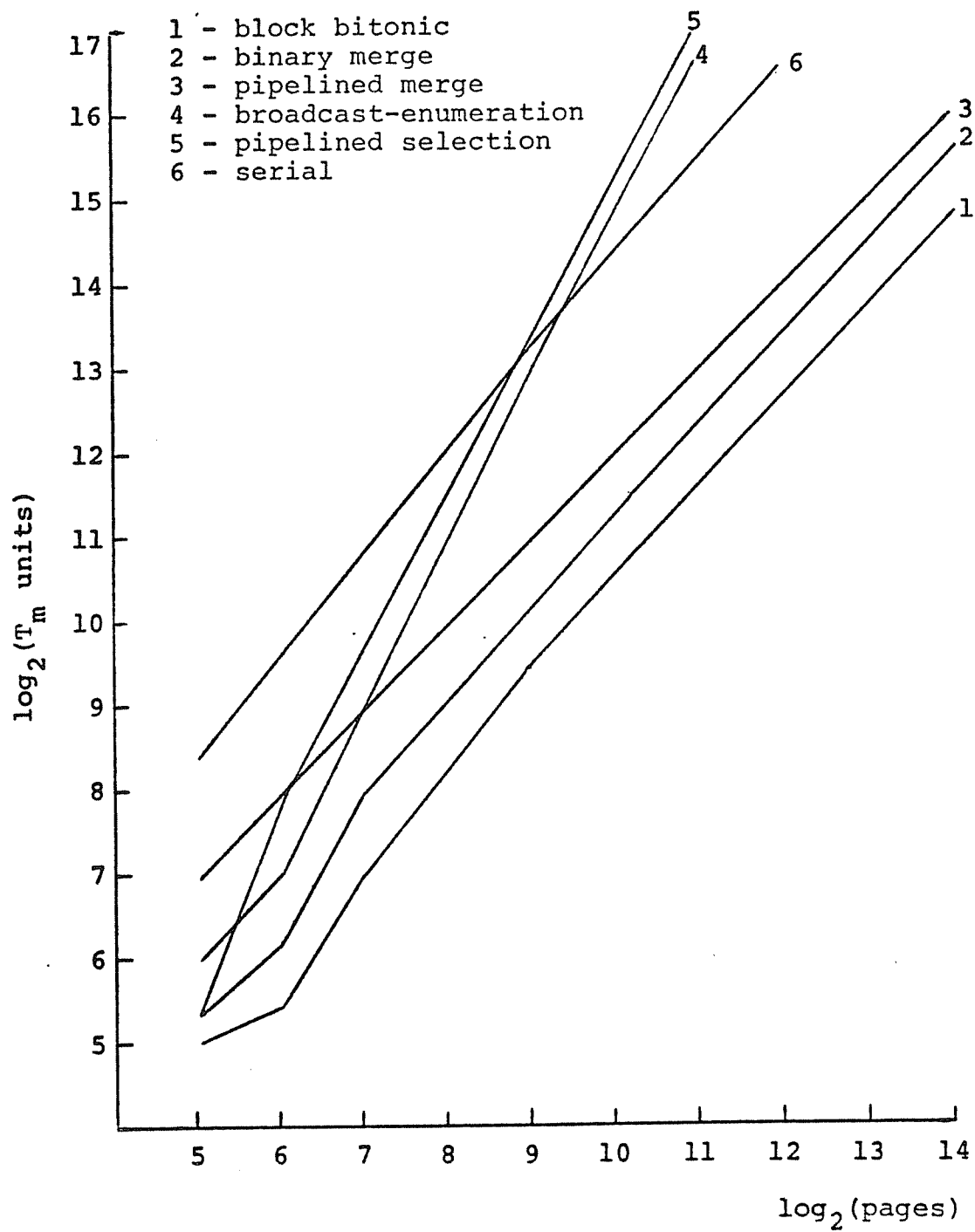


Figure 27. Computation time of the algorithms

In order to obtain a better insight on the relative performance of the algorithms (than what was shown by comparing their computation time), we have estimated the number of parallel page transfers along the links of the interconnection network and the total number of I/O transfers. In the following tables, we have summarized for several n/p ratios (number of pages/number of processors) the number of T_m (computation), T_{tr} (communication) and T_{io} (I/O) units required by each algorithm. First, we have fixed p (32), and considered several file sizes (256, 512, 1024 pages). Then, we have observed the impact of increasing the number of processors (from 32 to 128), for a fixed and large file size (8192 pages).

As indicated by the numbers in the second column of these tables, the communication cost of the first three algorithms is very low. In most cases, the number of page transfers is from 2 to 10 times less than the number of T_m computation units, and very significantly less than the number of I/O operations (in some cases 100 times less). On the other hand, for the other two algorithms (the pipelined selection and the broadcast-enumeration) the communication cost is very high. There are two reasons that explain the low communication cost of the first three algorithms. One is that during intermediate stages of these algorithms, output runs produced by a processor are written

Number of computation, communication and I/O units

Table 1 Number of pages:256, Number of processors: 32

Algorithm	Computation	Communication	I/O
pipelined merge	1012	134	1304
parallel binary merge	584	8	2048
block bitonic	314	168	11776
pipelined selection	2544	74	1920
broadcast enumeration	2064	128	4352
serial (2-way merge)	4096	0	4096

Table 2 Number of pages: 512, Number of processors: 32

Algorithm	Computation	Communication	I/O
pipelined merge	2035	263	5120
parallel binary merge	1224	259	5120
block bitonic	698	336	24576
pipelined selection	9184	4816	7936
broadcast enumeration	8224	16384	16896
serial (2-way merge)	9216	0	9216

Table 3 Number of pages: 1024, Number of processors: 32

Algorithm	Computation	Communication	I/O
pipelined merge	4082	520	11264
parallel binary merge	2568	515	12288
block bitonic	1498	672	51200
pipelined selection	34752	17824	32256
broadcast enumeration	32832	65536	66560
serial (2-way merge)	20480	0	20480

Sorting a larger file

Number of computation, communication and I/O units

Table 4 Number of pages: 8192, Number of processors: 32

Algorithm	Computation	Communication	I/O
pipelined merge	32751	4107	114688
parallel binary merge	23560	4099	147456
block bitonic	13786	5376	458752
pipelined selection	2113024	1060096	2093056
broadcast-enumeration	2097664	4194304	4202496
serial (2-way merge)	212992	0	212992

Table 5 Number of pages: 8192, Number of processors: 64

Algorithm	Computation	Communication	I/O
pipelined merge	32751	4107	114688
parallel binary merge	19466	4100	131072
block bitonic	8396	3584	557056
pipelined selection	1064704	536192	1044480
broadcast-enumeration	1048832	2097152	2105344
serial (2-way merge)	212992	0	212992

Table 6 Number of pages: 8192, Number of processors: 128

Algorithm	Computation	Communication	I/O
pipelined merge	32751	4107	114688
parallel binary merge	17676	4101	114688
block bitonic	5052	2304	671744
pipelined selection	540544	274240	520192
broadcast-enumeration	524416	1048576	1056768
serial (2-way merge)	212992	0	212992

(Note that the pipelined merge can only use 13 processors to sort a file of 8192 pages).

to disk (without processor to processor transfers). For example, both the binary merge and the block bitonic sort have a first stage (the "suboptimal stage"), during which the processors execute independently of each other a 2-way merge procedure and do not communicate. The second reason is that the algorithms take advantage of a high-bandwidth interconnection network by transferring many pages in parallel. If I/O transfers can also be performed in parallel, then the actual number of I/O transfers will be reduced by a factor equal to the degree of I/O parallelism.

Examination of Tables 1-6 reveals that for very large files the performance of the pipelined selection sort and the broadcast-enumeration sort is very poor relative to the performance of the first three algorithms. The number of communication and I/O page units are extremely high when the file size is 8192 pages. We have also observed that the computation cost was significantly higher for the pipelined binary merge than for the parallel binary merge and the block bitonic algorithms. Therefore, we shall now compare only these two algorithms.

A surprising property of the block bitonic sort is consistently indicated by every example presented in Tables 1-6. This algorithm requires a very large number of I/O transfers: about 4 to 5 times more than the binary merge and the pipelined merge. For 32 processors and up to 1024

pages, it requires more I/O than the pipelined selection algorithm. In addition, the number of I/O transfers it requires to sort a fixed size file increases with the number of processors. This behavior is illustrated by the last three tables ($n=8192$ and $p=32, 64$ and 128). We have also observed this same behavior for other values of n and p . Thus, unless I/O transfers can be performed with a high degree of parallelism, the parallel binary merge algorithm may perform better than the block bitonic sort.

4.7.3. Evaluation of I/O time

One question that naturally arises at this point is how strongly the I/O time required by the algorithms dominates the other two cost components (computation and communication). Obviously, the ratio between the I/O and the computation plus the communication times depends on the degree of I/O bandwidth that is available. In order to gain intuition on the ideal value for the I/O bandwidth, the following criterion can be used. The "ideal" bandwidth provides an average track transfer time $T(\text{ideal})$ such that:

$$(I/O \text{ units}) * T(\text{ideal}) \leq \\ (\text{Computation units} * T_m) + (\text{Communication units} * T_{tr})$$

We have estimated the value of $T(\text{ideal})$ for the (n,p) configurations that appear in the previous examples. A few of

the results that we have obtained are listed in Table 28.

For a single track transfer time of 16ms and an average access time of 16ms, these results indicate that, for the block bitonic sort, the 32 processors must be able to concurrently read or write 32 tracks. Thus, absolute synchronization of the I/O requests is necessary to insure a satisfactory performance from this algorithm. For the binary merge, on the other hand, less parallelization of the track transfers is required. However, since in this case the algorithm itself limits the amount of I/O parallelism (by requiring a single processor to perform a long sequence of write operations), it might not be possible to take advantage of a high degree of I/O bandwidth.

32 processors configuration:

n	T(ideal) binary merge	T(ideal) block bitonic
256	4.98	1.42
1024	10.06	1.52
8192	7.63	1.53

Figure 28. Ideal track transfer time in milliseconds

Worst case and best case execution time:

The ideal I/O bandwidth criterion provided some preliminary indication of the relative performance of the algorithms. It has shown, for example, that the block bitonic sort requires a much higher I/O bandwidth than the parallel binary merge sort to perform well. However, a more accurate comparison can be obtained by estimating the best and worst case execution time for each algorithm (Figure 29). These times are respectively obtained by using the best and worse I/O time formulas that were established for each algorithm (Section 4.3 and 4.4). Figure 30 shows that, when the best case I/O time is assumed, the block bitonic sort achieves the lowest execution time. With 32 processors, this algorithm can sort a large file approximately 10 times faster than a serial sorting algorithm. However, while the best I/O time obtained for the block bitonic sort is the lowest, the worst I/O time of this algorithm is very bad (it is only marginally better than the serial execution time). The parallel binary merge sort, on the other hand, provides a speedup of 5, even in the worst case.

4.8. Conclusion

Of the five algorithms that have been considered, the parallel binary merge sort appears to have the best overall performance. Depending on the merge pattern forced by the

36

L-

>-

.c

:l

l-

id

.

id

h

,,

c

r

o

,

t

y

n

,

e

l

e

3 - serial
1 - block bitonic
2 - binary merge

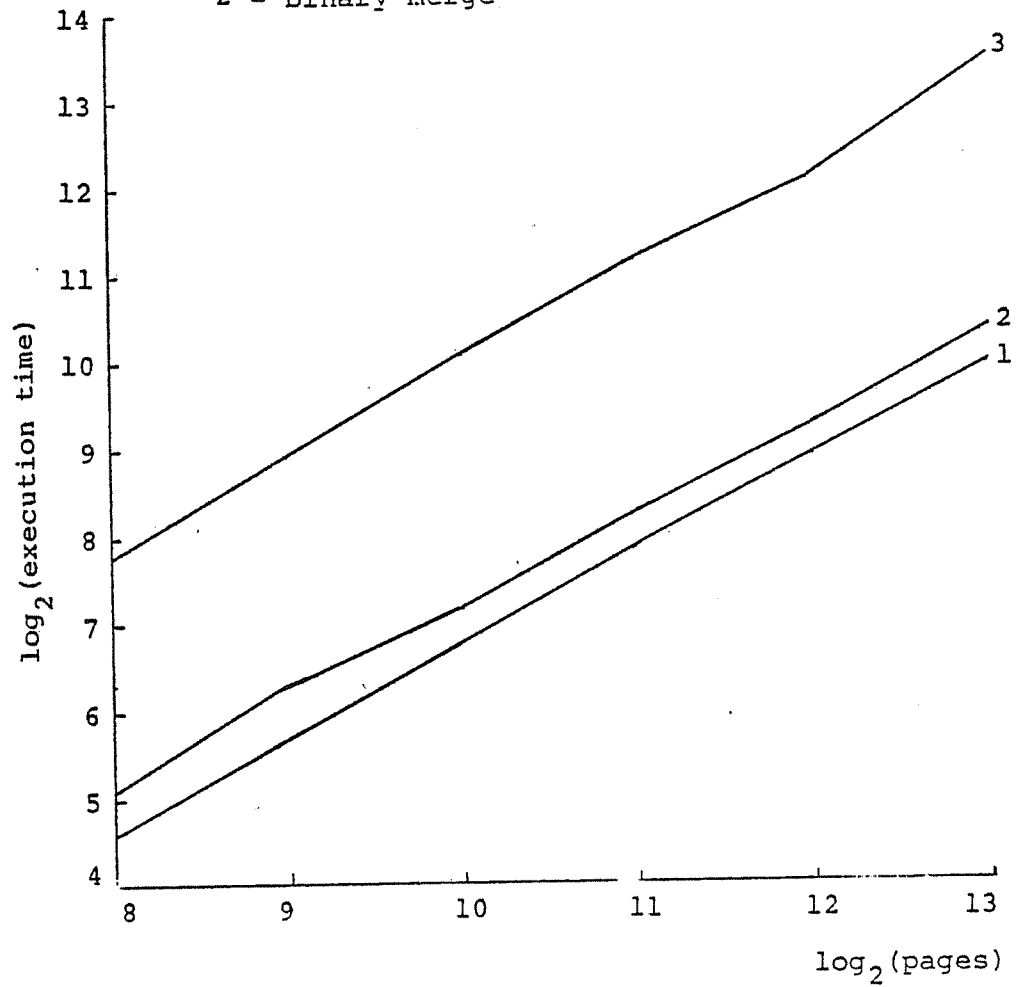


Figure 29. Best case execution time

3 - serial
2 - block bitonic
1 - binary merge

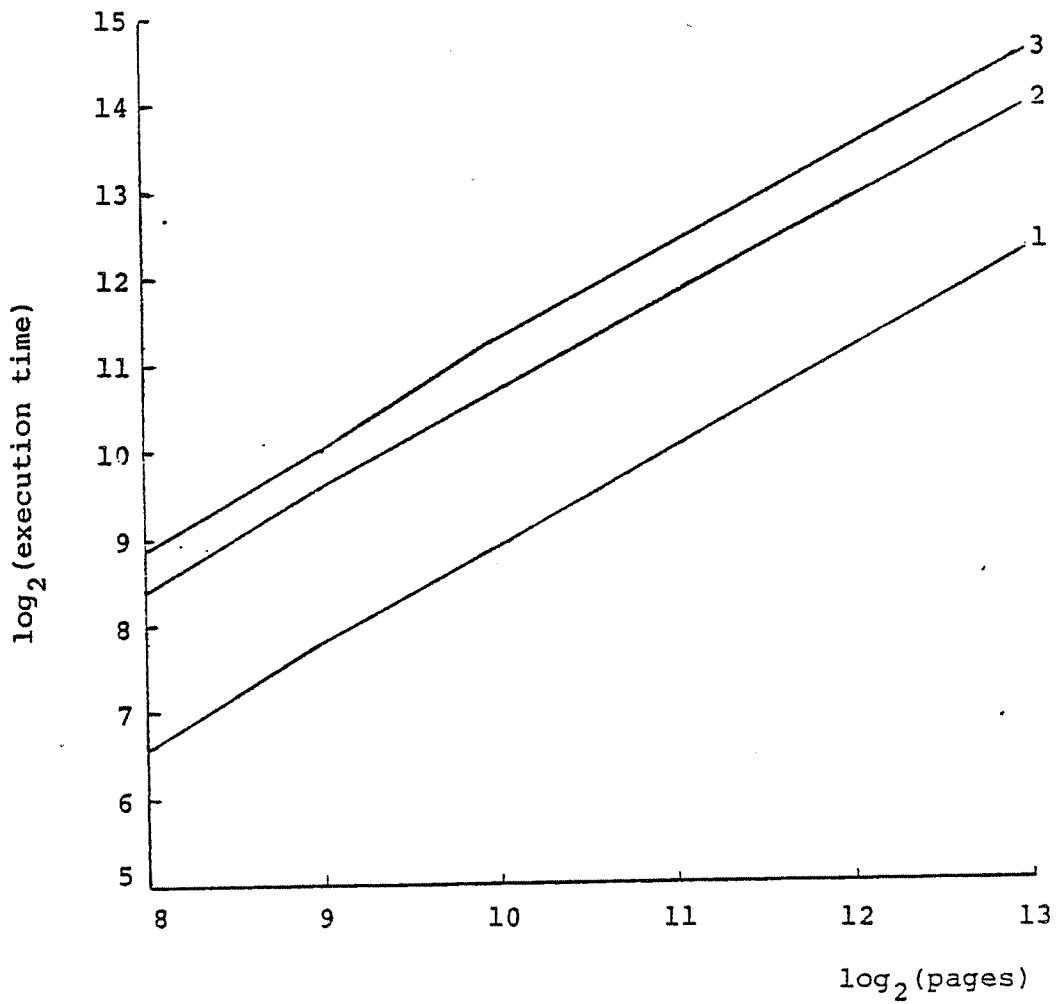


Figure 30. Worst case execution time

record values, the block bitonic sort can either perform extremely well or very poorly. Although the probability to hit the worst case performance is low (since it implies that each track transfer requires a long seek and a serial transfer), it must be taken into consideration. The pipelined merge sort is limited in the amount of parallelism it can use and cannot perform as well as the parallel binary merge. Finally, the other two algorithms (the pipelined selection sort and the broadcast-enumeration sort) perform poorly when the file is very large. However, for smaller files, their performance is comparable to the other algorithms. In this case, their simplicity might make them more attractive than the algorithms based on iterated merging.

By comparing the total execution time of the parallel external sorting algorithms and the serial 2-way external merge sort, we have obtained an indication of the parallel speedup that can be expected. The results indicated that by using 32 processors, the execution time of external sorting can only be reduced by a factor of 10. With the worst case assumptions (that is, when the merging pattern of every pair of input runs forces the track transfers to be serialized), the parallel speedup achieved is even lower.

Empirical measurements (if they ever become available) might be another way of evaluating parallel external sorting algorithms. For serial external sorting, numerous empirical studies have been done on real computers and real data. The results of these studies have complemented analytical results, and supplied valuable information on serial external sorting methods. In the near future, the high cost of external sorting will probably motivate the implementation of parallel external sorting algorithms. As in the case of serial external sorting, experiments will indicate new ways to improve and evaluate parallel external sorting techniques.

CHAPTER 5

DUPLICATE ELIMINATION

Files of data frequently contain duplicate entries and the decision whether and when to remove them must be made. For example, in relational database management systems (DBMSs) the semantics of the projection operator require that a relation be reduced to a vertical sub-relation and that any duplicates introduced as a side-effect be eliminated. In general, duplicate records may be introduced in a file either by performing an incorrect update operation or by being given a restricted view of the file. Identifying record fields such as names are often masked from an application program or from an output file before it is delivered to a user. In these cases, the amount of duplication can be significant and the cost of removing the duplicates substantial.

Duplicate elimination on a single processor is almost universally done by sorting. Because of the expense of sorting, relational DBMSs do not always eliminate duplicates when executing a projection. Rather, the duplicates are kept and carried along in subsequent operations. Only after the final operation in a query is performed, is the resultant relation sorted and duplicates eliminated.

The decision whether to eliminate duplicates or not (at various stages of the query execution) lies with the query optimizer subsystem of a DBMS. The purpose of a query optimizer is to schedule instructions from a query in a manner that will minimize the total query execution time. Typical factors affecting the decisions of an optimizer are: the types of operations in the query, the availability of auxiliary information about files (such as indices), the size of the input files, and the expected size of the intermediate and output files. For relational DBMSs, the expected size of intermediate relations is often kept in the form of "selectivity factors" which reflect the observed values of previously executed operations on the same relation.

To our knowledge, existing query optimizers do not adequately schedule duplicate elimination operations. The problem lies in the fact that the literature does not contain a model for analyzing the cost of this operation. In this chapter, we propose a combinatorial model for the use in the analysis of algorithms for duplicate elimination. We contend that this model can serve as a useful tool for a query optimizer to decide when to eliminate duplicates. We also describe a modified sorting method for eliminating duplicates and use our model to show its superiority over the accepted method of first sorting a relation and then

eliminating duplicates with a linear scan. These results are first presented for serial sorting. Then, the same model is used to evaluate a parallel duplicate elimination procedure.

In Section 5.1, we discuss particular aspects of duplicate elimination in relational DBMSs. We present three methods for performing duplicate elimination in Section 5.2. The rest of this chapter concentrates on a performance evaluation of one of these methods - a modified merge-sort procedure. In Section 5.3, we develop a combinatorial model that enables us to estimate the size of intermediate sorted runs produced by merging. In Section 5.4, we present some numerical evaluations based on this model. Our conclusions and suggestions for potential applications and extensions of our results are presented in Section 5.5.

5.1. Duplicate Elimination in a Relational DBMS

In relational database management systems, duplicate elimination constitutes a major part of the projection operation. Projecting a relation requires the execution of two distinct phases. First, the source relation must be reduced to a vertical subrelation by discarding all attributes other than the projection attributes. Then, duplicate tuples that may have been introduced as a result of

the first operation must be removed in order to produce a proper relation. The first operation, forming the projected tuples, can either be performed in a linear scan of the relation or may be performed in combination with an operation preceding the projection, in which case the cost of this step would be negligible.

For example, consider the "supply" relation:

supplier-no	part-no	source	destination	qty
-------------	---------	--------	-------------	-----

If we want to know which suppliers supply which parts in quantities larger than 1000 units, the relation must be restricted to (qty>1000) and projected on (supplier-no, part-no). Rather than creating a temporary relation for the restriction and then scanning it to discard the fields qty, source and destination, these fields should be eliminated as the restriction is executed. Since in the "supply" relation there may be many tuples with the same supplier-no and part-no attribute values, the result will be a list of non-unique two-attribute tuples. The second part of the projection consists of eliminating the duplicate tuples that are introduced by the first phase.

The amount of duplication introduced by the projection depends on the nature of the projection attribute(s). If we project on a primary key, then no duplicates will be

introduced.[1] On the other hand, if we discard a primary key and project on other attributes, a large amount of duplication may appear among the resulting tuples.

5.1.1. Implementation of the Projection Operation

Despite the fact that the duplicate elimination is an integral part of the projection operation as defined in Codd's relational algebra, relational database systems do not automatically implement it. Relations with duplicate tuples (which are not proper relations according to the relational algebra semantics) are in fact operated on by restrictions, joins, and other relational operators. Because duplicate elimination is expensive and because the tradeoffs between performing it or putting up with some inconsistent and redundant data are not clear, most database systems implementations (including relational systems such as System R [ASTR76] and INGRES [STON76]) postpone it to the very last stage of query processing. At that stage, the result tuples are sorted and duplicates are eliminated. If duplicate elimination is not systematically performed with every projection, the sets operated on by the relational operators such as selection and joins are not relations, and the database management system does not really

[1] A primary key is an attribute or a set of attributes which uniquely identifies a tuple.

conform to the relational model. Katz and Goodman [KATZ81] are currently investigating an extension of the relational model that allows for duplicates in relations. This extension deals with multisets that are viewed as a generalization of relations.[2] [KATZ81] shows how the relational algebra operators selection, join, and projection can be generalized to multisets, and introduces a new operator to explicitly specify duplicate elimination. Our study does not deal with the semantics of duplicate elimination. Its goal is to develop tools for implementing and evaluating the cost of this operation, whether or not it is done in the context of a relational database management system. In the case of a relational database, establishing an accurate cost formula for the projection can make improved query processing strategies possible. The first step in establishing such a formula is to evaluate the number of tuples in the projection.

5.1.2. Size of the Projected Relation

It is usually assumed that the database system dictionary can supply a reliable estimate of the size of a relation projected on any specified subset of attributes. This estimate may be based on an "a priori" knowledge about

[2] [KNUT73] defines a multiset as a set of non-unique elements. In Section 5.3, we define more specifically multisets that are relevant to our study.

the number of distinct values that the projection attribute(s) can take on. It is reasonable to assume that this kind of information would be stored for all permanent relations, since it indicates the cardinality of the attributes domains. However, the same information will neither be readily available for temporary relations created during intermediate stages of a query execution nor be inexpensive to compute. In the event that the size of a projected relation must be quickly estimated, a reasonable approximation may be achieved by assuming that a relation size is proportional to its tuple length [KERS80]. Let $|R|$ and $|R_p|$ denote the number of tuples in the source relation and in the projected relation, respectively. Then,

$$|R_p| = f_p * |R|$$

where the "projectivity" f_p equals the tuple length in bytes divided by the length of the projection attribute(s) in bytes. When indices on the projection attributes exist, the size of the projection can be estimated as

$$|R| * \prod_{a_j \in A} (1/s_j)$$

where A is the set of the projection attributes and s_j is the index selectivity for attribute a_j [YAO79].

Assuming that we have a reliable estimate for the size of the source relation and the size of its projection, the

cost of the projection can be essentially defined as the cost of eliminating duplicates in a multiset of records, knowing the size of the multiset and the number of distinct records in it.

5.2. Algorithms for Duplicate Elimination

Using any sorting method with the entire record taken as the comparison key will bring identical records together. Since many fast sorting algorithms are known, sorting appears to be a reasonable method for eliminating duplicate records. This section briefly describes three methods for duplicate elimination, two of which are based on sorting. For the sake of simplicity, only the serial version of these methods will be described. However, each of them can be extended to a parallel method for duplicate elimination. The first method is an external 2-way merge-sort followed by a scan that removes the duplicates. A parallel version of this method that will be investigated is based on the parallel binary merge sort (Section 4.3). The second method is a modified version of an external 2-way merge-sort, which gradually removes duplicates as they are encountered. The third method consists of using an auxiliary bit array that is obtained by hashing the record fields. This method was introduced by [BABB79], for efficiently realizing the relational join and projection operations. We discuss it for the sake of completeness, but we

do not compare it with the other methods as it requires the use of specialized hardware for efficient operation.

We assume that the file resides on a mass storage device such as a magnetic disk (although for very large files, magnetic tape may be used as the storage media). It consists of fixed size records that are not unique. The amount of duplication is measured by the "duplication factor" f which indicates how many duplicates of each record appear in the file, on the average. The records are grouped into fixed size pages. An I/O operation transfers an entire page from disk storage to the processor's memory or from memory to disk. The file spans N pages, where N can be arbitrarily large, but only a few pages can fit in the processor's memory. The cost of processing a complex operation such as sorting or duplicate elimination can be measured in terms of page I/O's because I/O activity dominates computation time for this kind of operations.[3]

5.2.1. The Traditional Method

For a large data file, duplicates are usually eliminated by performing an external merge-sort and then scanning the sorted file. Identical records are clustered

[3] In fact, since for algorithms such as merge-sort the sequence of pages to be read is known in advance, pages can be prefetched enabling computation time to be completely overlapped with I/O time.

together by the sort operation, therefore they are easily located and removed in a linear scan of the sorted file. We assume that the processor's memory size is about three pages and some working space. In this case, the file can be sorted using an external 2-way merge sort (Section 4.1.1). A 2-way merge procedure requires $\log_2 N$ phases with N page reads and N page writes at each phase (since the entire file is read and written at each phase).

After the file has been sorted, duplicate elimination is performed by reading sorted pages one at a time and copying them in a condensed form (i.e. without duplicates) to an output buffer. Again an output buffer is written to disk only after it has been filled, except for the last buffer which may not be filled. Thus the number of page I/O's required for this stage is:

$$N \text{ (reads)} + \text{ceil}(N/f) \text{ (writes)}$$

The total cost for duplicate elimination measured in terms of I/O operations is:

$$2N \log_2 N + N + \text{ceil}(N/f)$$

5.2.2. The Modified Merge-sort Method

Most sorting methods can be adapted to eliminate duplicates gradually. [MUNR76] establishes a computational bound for the number of comparisons required to sort a multiset, when duplicates are discarded as they are encoun-

tered. Since we are dealing with large mass storage files, we are solely interested in working with an external sorting method. A two-way merge-sort procedure can be easily modified to perform a gradual elimination of duplicates. If 2 input runs are free of duplicates, then the output run produced by merging them should retain only one copy of each record that appears in both input runs (see Figure 31). Whenever two input tuples are compared and found to be identical, only one of them is written to the output run and the other is discarded (by simply advancing the appropriate pointer to the next tuple). The cost of the duplicate elimination process using this modified merge-sort is then determined by two factors: the number of phases and the size of the output runs produced at each phase.

Number of Phases

The number of phases required to sort a file with gradual duplicate elimination is the same regardless of the number of duplicate tuples. That is, $\log_2 n$ merge phases are required to sort a file of n records. This is true even in the extreme case when all the file records are identical. In this particular case, the run size is always 1 and every merge operation consists of collapsing a pair of identical elements into one element. By the same argument, if we start an external merge-sort with N internally

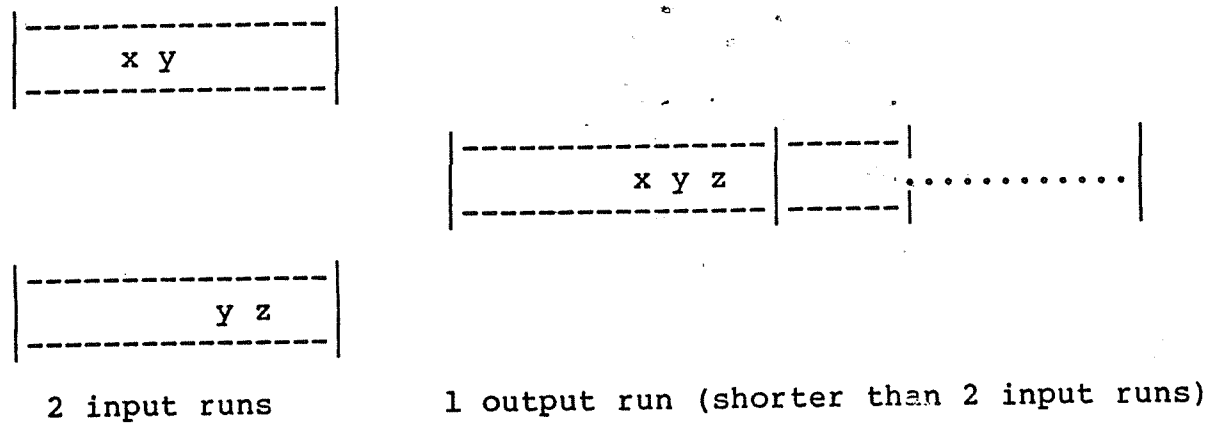


Figure 31. Modified Merge

sorted pages, the number of phases required is $\log_2 N$, whether or not duplicates are eliminated.

Size of Output Runs

Since we know the number of phases, the number of I/O operations required to execute the modified merge-sort will be completely determined if we have a method to measure how the runs grow as the modified merge-sort algorithm proceeds. When a two-way merge-sort is performed, the size of the runs grows by a factor of 2 at each step. However, if the merge procedure is modified in order to eliminate duplicates as they are encountered, the size of the runs does not grow according to this regular pattern. Suppose that the modified merge-sort procedure is executed without

throwing away the duplicates as they are encountered. Instead, the duplicates would only be marked so that at any step of the algorithm they can be rapidly identified. Then, the size of an output run produced at phase i would still be 2^i but the number of distinct elements in the run would only be equal to the number of unmarked elements. Thus, it seems that a reasonable estimate for the average size of a run produced at the i th phase of a modified merge procedure is the expected number of distinct elements in a random subset of size 2^i of a multiset. In Section 5.3, we present a combinatorial model that provides us with such an estimate.

5.2.3. The Hashing Method

Essentially, this method works as follows. A bit array $(M(I), I:1..k)$ with about as many entries as the number of distinct records is used to check for duplication. Rather than comparing the records themselves, a hash function provides a way to establish if 2 records are identical. Initially, all the entries in M are set to zero. Then, for each record read, all the fields are concatenated and the resulting string is hashed to provide the appropriate index, say I . If $M(I)=0$, the current record is written to the output list and $M(I)$ is set to 1. This procedure ensures that no record is written twice to the output list, since identical records must hash to the same

address. However, some records may be left out only because they "collide" with previously read records. Thus, each collision means losing a "good" record. For this reason, it is very important to minimize the number of collisions. One way to achieve this goal is to increase the size of the bit array ([BABB79] recommends 4 times as many entries as the number of distinct records). A further improvement can be achieved by using several hash functions rather than a single one. The source file is scanned once for each hash function, and an output file (of non-colliding records) is created for each scan. Then the union of the output files is taken as the result file. The probability of missing records can be substantially reduced by using several statistically independent hash functions, since it is unlikely that different records will collide for each of these functions.

[BABB79] shows that the hashing method can be very fast, when specialized hardware is used. The main problem remains the probability of missing any records. If there is not enough a priori knowledge on the data in order to determine that the expected number of missing records will be extremely small [4], or if no chance to miss a record can be taken, the cost of performing duplicate elimination with this method becomes extremely high (since it would require scanning the source file again to check if no

record has been left out of the output file).

On a conventional computer, it seems that any duplicate elimination method not based on sorting would require an exhaustive comparison of all records and therefore lead to a slower performance. However, parallel processing may offer other alternatives. Various parallel architectures and algorithms are investigated for the elimination of duplicates in two recent studies ([BORA80], [GOOD80]). It may be the case that features such as broadcast of data to several processors will be the source for faster algorithms for duplicate elimination. The results that have been obtained in this area are not conclusive and will not be presented as they are beyond the scope of this study.

5.3. A Combinatorial Model for Duplicate Elimination

In this section, we consider the problem of finding all the distinct elements in a multiset. A multiset is a set $\{x_1, x_2, \dots, x_n\}$ of not necessarily distinct elements. We assume that any two of these elements can be compared yielding $x_i > x_j$, $x_i = x_j$ or $x_i < x_j$. The x_i 's may be real numbers or alphanumeric strings that can be compared according to the lexicographic order. Or they may be

[4] For 50K distinct records, [BABB79] estimates that a bit array of 1M bits and 4 scans can reduce the error rate to 10^{-11} .

records with multiple alphanumeric fields, with one (or a subset of fields) used as the comparison key. The elements in the multiset are duplicated according to some distribution f_1, f_2, \dots, f_m . That is there are f_1 elements with a "value" v_1 , f_2 elements with a value v_2, \dots, f_m elements with a value v_m , and $\sum f_i = n$. When n is large and the values are uniformly distributed, we may assume that

$$f_1 = f_2 = \dots = f_m = f$$

and therefore

$$n = f \cdot m$$

In this case, we define f as the "duplication factor" of the multiset.

5.3.1. Combinations of a Multiset

Consider the following problem. Suppose we have a multiset of n elements with a duplication factor of f and m distinct elements so that $n=f \cdot m$. Let k be any integer less or equal to n . How many distinct combinations of k elements can be formed where all the m distinct elements appear at least once? This number is denoted by $C_{fm}(k)$. We consider combinations rather than arrangements because we are interested in the identity of the elements in a subset but not in their ordering within the subset. The notation $\binom{p}{q}$ is used to represent a q -combination of p distinct elements, with the convention $\binom{p}{q}=0$ for $q>p$.

Lemma 1:

$$c_{fm}(k) = \binom{f+m}{k} - \binom{m}{1} \binom{f+(m-1)}{k} + \binom{m}{2} \binom{f+(m-2)}{k} - \dots + (-1)^{m-1} \binom{m}{m-1} \binom{f}{k}$$

Proof: The intuitive meaning of lemma 1 is that the number of combinations with exactly m distinct elements is equal to the number of combinations with at most m distinct elements minus the number of combinations with $m-1, m-2, \dots, 1$ distinct elements. To prove the lemma, we express the total number of combinations of size k in terms of the number of combinations of size k with m distinct elements, of the number of combinations of size k with $m-1$ distinct elements, etc.

$$\binom{f+m}{k} = c_{fm}(k) + \binom{m}{1} c_{f(m-1)}(k) + \binom{m}{2} c_{f(m-2)}(k) + \dots$$

$$\binom{f+(m-1)}{k} = c_{f(m-1)}(k) + \binom{m-1}{1} c_{f(m-2)}(k) + \binom{m-1}{2} c_{f(m-3)}(k) + \dots$$

...

By combining these expressions to form the right-hand side sum in lemma 1, all the $c(k)$ cancel each other except for $c_{fm}(k)$. Notice also that k might be greater than $f(m-i)$ for some $i > 0$ which according to our notation, would imply that some of the terms in the right hand side sum become zero.

5.3.2. The Average Number of Distinct Elements

Starting with a multiset that has m distinct values and a duplication factor f , there are $\binom{fm}{k}$ subsets of size k . Thus, the probability that a random subset of size k contains exactly d specific distinct elements ($d \leq m$) is equal to:

$$c_{fd}(k) / \binom{fm}{k}$$

The expected number of distinct elements in a random subset of size k can be computed by averaging over all possible values of d . The lowest possible value of d is $\lceil k/f \rceil$ since d distinct elements cannot generate a set larger than $f*d$. On the other hand, there can be at most m distinct values since we are considering subsets of a multiset with m distinct values.

Lemma 2: The expected number of distinct elements in a subset of size k is:

$$\text{av}_{fm}(k) = \left\{ \sum_{d=\lceil k/f \rceil}^{\min(k,m)} d \cdot \binom{m}{d} \cdot c_{fd}(k) \right\} / \binom{fm}{k}$$

Lemma 3: For $i > 1$

$$(m-i) - (m-i+1) \cdot \binom{i}{1} + (m-i+2) \cdot \binom{i}{2} - \dots = 0$$

Proof: Let us consider the product $x^{m-i}(1-x)^i$. By expanding the second factor, we have

$$x^{m-i}(1-x)^i = x^{m-i} - \binom{i}{1} x^{m-i+1} + \binom{i}{2} x^{m-i+2} - \dots$$

and

$$\frac{d}{dx} \{ x^{m-i} (1-x)^i \} = (m-i)x^{m-i-1} - (m-i+1) \binom{i}{1} x^{m-i} + (m-i+2) \binom{i}{2} x^{m-i+1} - \dots$$

For $x=1$ and $i>1$ this derivative is equal to zero.

Lemma 4: For $k \geq m$

$$\sum_{d=\lceil k/f \rceil}^k d \binom{m}{d} * c_{fd}(k) = m \binom{f^*m}{k} - m \binom{f^{(m-1)}}{k}$$

Proof: Let

$$S = \sum_{d=\lceil k/f \rceil}^m d * \binom{m}{d} * c_{fd}(k)$$

Since for each k , $c_{fd}(k)$ is a linear combination of terms of the form $\binom{f^{(m-i)}}{k}$ and since the upper bound for d in S is m , S may be rewritten backwards as a linear combination of terms of the form $\binom{f^{(m-j)}}{k}$, $j=0,1,\dots$. Then the coefficient of $\binom{f^m}{k}$ is $m \binom{m}{m}$. The coefficient of $\binom{f^{(m-1)}}{k}$ is $(m-1) \binom{m}{1} - m^2 = -m$.

For $i>1$, the coefficient of $\binom{f^{(m-i)}}{d}$ is:

$$(m-i) - (m-i+1) \binom{i}{1} + (m-i+2) \binom{i}{2} - \dots$$

which is null by lemma 3.

Theorem 1: If $k \geq m$

$$av_{fm}(k) = m - m * \left\{ \binom{f^{(m-1)}}{k} / \binom{f^*m}{k} \right\}$$

It is interesting to notice that when f is large (that is the duplication factor is high), $av_{fm}(k)$ becomes a function of m and k only. This can be proven as follows:

$$\binom{f(m-1)}{k} / \binom{f*m}{k} = \prod_{i=1}^k (f*m-f-k+i)/(f*m-k+i)$$

which is approximately equal to $(m-1/m)^k$ for large f . Therefore, $av_{fm}(k) \approx m(1 - (m-1/m)^k)$ for large m .

This result confirms the intuitive idea that the number of distinct elements in a random subset of a large multiset depends only on the size of the subset and on the number of distinct values in the multiset.

For smaller duplication factors, as we keep f and m constant, $av_{fm}(k)$ increases monotonically as a function of k , until for some $k=k_0$ it reaches the value m . From then it remains constant as k increases from k_0 to $f*m$. Figure 32 displays the function $av_{fm}(k)$ for $f*m=32768$ and for three different values of f (8, 16, 32). The value k_0 is of particular interest. It indicates how large a random subset of a multiset must be in order to contain at least one copy of all the distinct elements.

It is interesting to note that the problem we address here is somehow related to the classical "occupancy problem" of the Bose-Einstein statistics [FELL68]. The n ele-

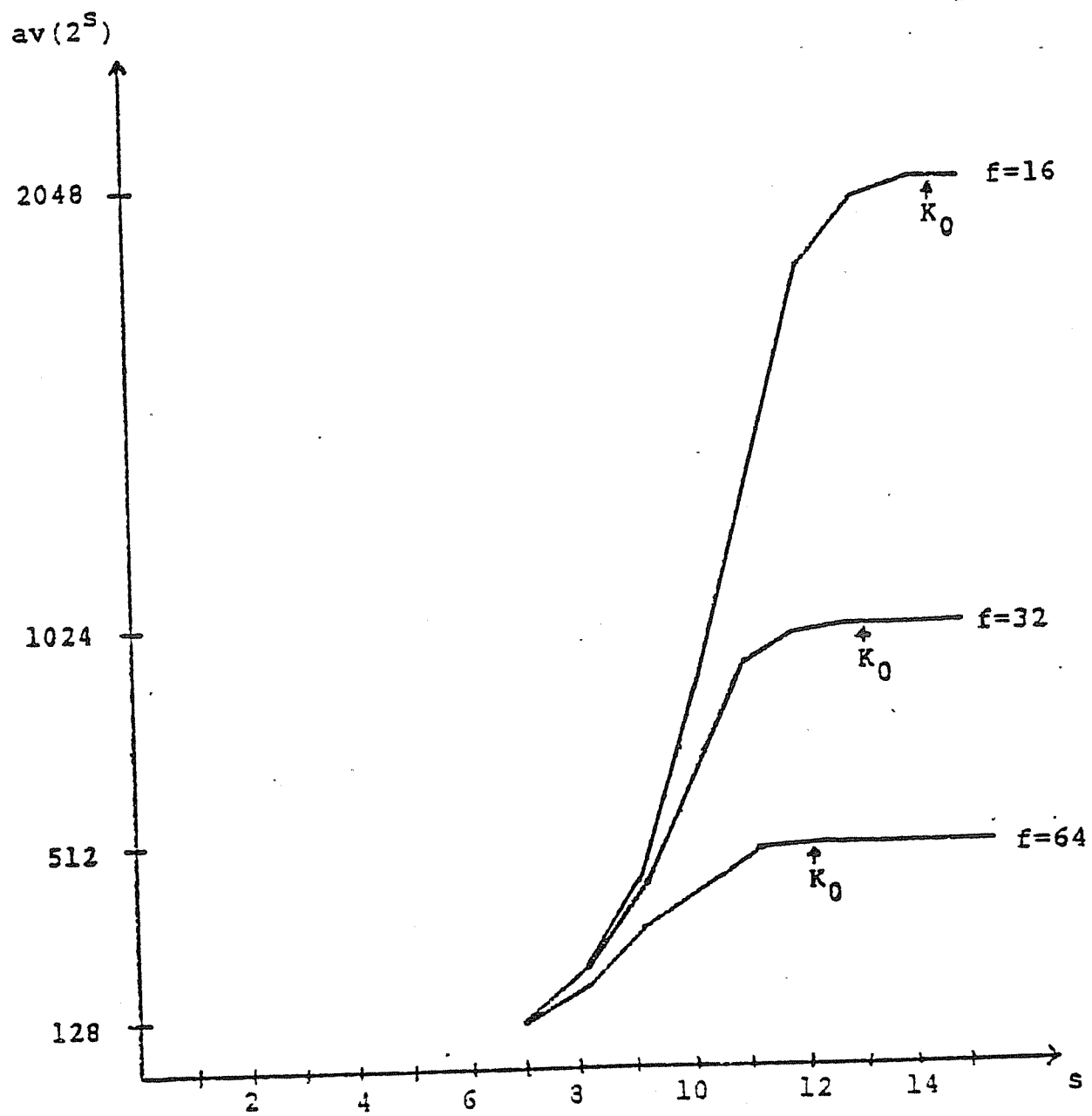


Figure 32. Number of Distinct Records in Successive Runs

ments of a multiset may be identified with n particles and the m distinct elements in the multiset may be pictured by m distinct cells in which the particles can fall. All the duplicates of one element are then represented by the set of particles that fell into a single cell. Assuming that all elements are equally duplicated, i.e. $n=f \cdot m$ and that there are f copies of each element, is equivalent to assuming that all cells end up containing the same number of particles. In fact, this is only one of many possible states and a more accurate modeling of duplication should consider the number of particles in one cell as a random variable with mean equal to $f=n/m$. It is known that for large n and m , the distribution of this variable is a Poisson distribution with mean n/m [FELL68]. Estimating the number of distinct records in a multiset is similar to estimating the number of occupied cells. Here again, it has been shown that the number of empty cells (that is m - the number of occupied cells) is a random variable with a Poisson distribution. Thus, for very large duplication factors, the probability of finding a specified number of distinct records in a subset of records can be estimated using the tools developed for the classical occupancy problem.

5.4. Cost Analysis of Duplicate Elimination

As discussed in Section 5.2, the cost of a modified merge sort is completely determined if the size of intermediate output runs can be estimated. In this section, we evaluate this cost and compare it to the cost of a traditional merge-sort. We then use a very similar technique to evaluate the cost of a modified parallel binary merge sort. We assume that the source file consists of n non-unique records, with a duplication factor f . The modified merge algorithm will produce an output file of $m=n/f$ distinct records. Both the source file and the output file records are grouped into pages and all pages, except possibly one, contain the same number of records ("page size" below). The cost of duplicate elimination is measured by the number of pages read and written, assuming that the main memory can fit no more than two page input buffers and one page output buffer. When an intermediate run is produced, records are also grouped into full pages before any output buffer is written out. Only the last page of a run may not be full. Therefore, the number of pages written when an output run is produced will be:

$\text{ceil}(\text{number of records in a run} / \text{page size}).$

5.4.1. Cost of serial duplicate elimination

We assume that the external merge procedure starts with internally sorted pages, and that each of these pages is free of duplicates. This assumption is legitimate if the records are uniformly distributed across pages in the source file, and if the number of distinct records is much larger than the number of records that a single page can hold.

If there were no duplicates, the number of records in each input run read at phase i would be 2^{i-1} times the page size, since the merge procedure is started with runs that are one page long. Similarly, the number of records in an output run produced at phase i would be 2^i times the page size. Suppose the page size (measured in number of records that a page can hold) is p . Therefore, when duplicates are gradually eliminated, the expected number of records in an input run read at phase i is equal to $av_{fm}(2^{i-1}p)$, and the expected number of records in an output run produced at phase i is equal to $av_{fm}(2^i p)$, using the notation defined in Section 5.3. On the other hand, the number of runs produced at phase i is $n/2^i p$ (where $n=fm$ is the number of records in the source file). Therefore, the number of pages read at phase i is

$$2 * \text{ceil}[av_{fm}(2^{i-1}p)/p] * n/2^i p$$

and the number of pages written is

$$\text{ceil}[\text{av}(2^i p) / p] * n / 2^{i * p}$$

Using these formulae, we have summarized in Figure 33 the total number of page I/O's required to eliminate duplicates from a file of 131072 records. With 128 records per page, this file spans 1024 disk pages. We have considered various duplication factors from 2 (i.e. there are 2 copies of each record) to 64 (64 copies of each record) The results indicate that a modified merge sort requires substantially fewer page I/O's than a standard merge sort, especially when the amount of duplication is large. When a standard merge sort is used to eliminate duplicates, it must be followed by a linear scan of the sorted file. Therefore, we also show this augmented cost in the rightmost column of the table in Figure 33.

A further reduction of page I/O's can be achieved by terminating the modified merge procedure as soon as the runs have achieved the result file size. When this happens, all the output runs will essentially be identical and each of them contains all the distinct records. As we observed in Section 5.3, this may occur a few phases before the final phase, e.g. at phase number $(\log_2 N) - i$, for some $i > 1$ (N being the number of pages spanned by the source file). When this phase is reached, a single run may be taken as

f	modified merge	standard merge	std merge+scan
2	19008	20480	22046
4	17400	20480	21760
8	15664	20480	21632
16	13840	20480	21568
32	12000	20480	21536
64	10192	20480	21520

Figure 33. Cost of serial duplicate elimination

the result file since it contains all the distinct records and no duplicates. Therefore, the elimination process is complete and one may save the additional I/O operations which serve only to collapse together several identical runs. Figure 34 shows the I/O cost of this "shortened" procedure, compared to the cost of a complete modified merge sort: For this file size, the savings in page I/O's can reach up to 7% of the total cost. For a smaller file size (32K records) and a small duplication factor, we have observed an improvement of the order of 10%. When varying the file size and the duplication factor, we have observed that the improvement was greater for very small or very

large duplication factors, while it was smaller in the mid-range values (e.g $f=8$ and $f=16$).

5.4.2. Cost of parallel duplicate elimination

A parallel binary merge sort (Section 4.3) can be modified, like the serial 2-way merge, in order to perform gradually the duplicate elimination. During the suboptimal stage, each processor independently executes the first phases of a serial 2-way merge. Thus, during this stage, the number of I/O operations is the same as the number of I/O operations in the serial algorithm divided by the number of processors (assuming that the I/O transfers can

<u>f</u>	<u>modified merge</u>	<u>shorter merge</u>	<u>improvement</u>
2	19008	17728	1280
4	17400	16264	1136
8	15664	15280	384
16	13840	13264	576
32	12000	11328	672
64	10192	9472	720

Figure 34. Early termination of modified merge

be done in parallel). After the suboptimal stage, additional I/O operations are needed only for reading the file in parallel once (during the optimal stage), and for writing the result file. Writing the result file is done serially by the root processor. In the case of parallel sorting without duplicate elimination, this last operation was very slow. However, in the case of a modified sort, the result file can be very small compared to the source file (if the duplication factor is high). Thus, the cost of the serial write is not as prohibitive. Figure 35 summarizes the cost of parallel duplicate elimination for a binary tree configuration with 16 leaf processors. The same assumptions as before were made about the size and the structure of the file.

5.4.3. Non-Uniform Duplication

Since we have only estimated the expected size of the runs, our numbers are only accurate provided that the actual run size does not fall too far away from that average. This will certainly not happen if the records are uniformly distributed in the source file. Finally, it is very important to note that if there is no a priori information about the number of duplicate records present in the source file, the modified sort-merge can still be used to eliminate duplicates and the procedure can be terminated as soon as the run size stop growing. When this condition is

f	serial modified merge	modified parallel merge
2	19008	1278
4	17400	1015
8	15664	872
16	13840	778
32	12000	696
64	10192	608

Figure 35. Cost of parallel duplicate elimination

verified, a single run can be taken as the result file, although a precise statement about the probability that such a run indeed contains all the distinct records requires a more elaborate statistical model than the one we have used.

5.5. Conclusion

In this chapter, we have presented a model for evaluating the cost of duplicate elimination. We have shown how, by modifying a 2-way merge-sort, duplicates can be gradually eliminated from a large file at a cost which is substantially less than the cost of sorting. Accurate

formulas have been established for the number of disk transfers required to eliminate duplicates from a mass storage file, first with a serial algorithm, then with a parallel algorithm. These formulas can be used whenever there exists an a priori estimate for the amount of duplication present in the file. When such an estimate is not available, it is argued that the modified merge-sort method (serial or parallel, depending on the context) should still be used. In this case, a condition for testing that all duplicates have been removed is described.

We have based our analysis on a combinatorial model that characterizes random subsets of multisets. Only a particular category of multisets has been considered, where all elements have the same order. Thus, our results are only accurate for files with a uniform duplication factor (i.e. each record is replicated the same number of times in the entire file). Refining our analysis would require the use of more sophisticated statistical tools to model more accurately the distribution of duplicates. However, for files with a large number of records and with many duplicates, our model would provide a reasonable approximation.

In addition to generalizing our cost evaluation model to the case where the records are not uniformly duplicated, it would be of interest to model the cost of duplicate elimination on parallel computers, using methods other than

parallel sorting. As mentioned in Section 5.2, algorithms for performing this operation have already been developed ([BORA80], [GOOD80]). However, measuring the execution time of these algorithms requires the need of a different cost model.

The motivation for our work was the need for a method to evaluate the cost of duplicate elimination. To our knowledge most query optimizers in relational DBMSs schedule a duplicate elimination operation in an ad hoc manner. The model developed in this paper can serve as a tool to be used by a query optimizer in estimating the cost of eliminating duplicates from a relation. Using this estimated cost an optimizer can schedule operations so that the total execution time of the query is minimized.

CHAPTER 6

CONCLUSION

In this final chapter, we summarize the contributions of this dissertation and point to areas for future research.

6.1. Contributions

While an extensive literature exists that addresses computation and communication issues in parallel processing, until now, the impact of I/O on the performance of parallel algorithms has not received adequate consideration. In particular, the problem of parallel external sorting has not been previously addressed. In this thesis, we have demonstrated that parallelism can be employed to efficiently sort very large files. We have proposed several new algorithms for parallel external sorting, and estimated their execution time. By assuming that several processors shared access to a modified, moving-head disk device, we have demonstrated that external sorting can be performed faster than by a conventional processor. However, the parallel speedup achieved is not as high as the speedup provided by fast parallel internal sorting algorithms.

In the context of parallel external sorting, we have considered two aspects of parallel processing that had not previously been considered. One is the implementation and the feasibility of parallel algorithms, within the limits imposed by current technology. The other is the development of a comprehensive cost analysis framework for those parallel algorithms that perform I/O intensive tasks.

In our performance evaluation of parallel external algorithms, we have accounted for three major cost components: computation, communication and I/O. While estimating the computation and the communication times did not require the development of new techniques, modeling I/O time was difficult. Several criteria have been proposed to estimate the disk access time and the number of page transfers required by a parallel external sorting algorithm.

As a major application of parallel external sorting, we have considered its use as a tool for the execution of complex database operations. In particular, we proposed to perform the duplicate elimination operation by using a modified merge-sort algorithm. A new combinatorial model has been developed, that provides an accurate estimate for the cost of this operation (both in the serial and the parallel cases).

6.2. Future research

There are two main research directions that we plan to pursue. One is to investigate applications of parallel sorting in the area of database management systems. The other is to refine our methodology for the analysis of parallel algorithms. In this section, we point at specific problems in these two directions that we intend to study in the near future.

6.2.1. Parallel sorting in database machines

In recent years, specialized hardware has been proposed to enhance the performance of DBMS's. Several basic architectures have been proposed for "database machines" that unlike traditional computers, are designed with database applications in mind. Support of massive parallelism has been investigated in most database machines designs ([Ozka75], [Lin76], [Su75], [Bane78], [DeWi79]). However, research on database machines has, for the most part, been architecture oriented. As a result, parallel algorithms for database operations have not received much consideration, and performance evaluation tools have rarely been investigated [DeWi81]. Although an extensive literature exists on parallel computation and parallel sorting, it does not address the problems specific to database management, namely the necessity of dealing with very large

volumes of data.

The join operation is one of the most time-consuming operations in a relational database management system. Thus, many algorithms have been investigated for its realization. In conventional systems, a variety of methods are used, depending on the availability of auxiliary structures. In the general case, sorting has been shown to be the best approach [Blasg77]. For database machines that use parallel processors, both indexing [Hsiao79] and hashing [Good80] have been proposed as building blocks for a fast realization of the join. In the case that an efficient broadcast facility is available, a simple nested-loops algorithm can achieve a relatively high performance when one of the operand relations is not too large [Bora81].

A parallel join of two relations can be performed by first doing a parallel sort on both relations to be joined (assuming that they are not both already sorted on the join attribute). After both relations have been sorted, they are joined by a single processor. Since the relations have been sorted, the complexity of the join step is the cost of merging two sorted files. Let n and m be the sizes in pages of the R and T relations, respectively. Also, let S_j be the join selectivity factor. Then, for this algorithm, the time to perform the join of R and T is equal to

$$\begin{aligned}
T &= T(\text{sort } R) + T(\text{sort } T) + \\
&\quad + T(\text{merge 2 sorted files}) \\
&= T_{\text{sort}}(n) + T_{\text{sort}}(m) + \\
&\quad + (n+m) * (T_{\text{acc}} + T_r) \quad (\text{read sorted relation}) \\
&\quad + 2 * \max(n, m) * T_m \quad (\text{join } n \text{ to } m \text{ pages}) \\
&\quad + m * n * S_j * (T_{\text{sk}} + T_{\text{tr}}) \quad (\text{write result relation})
\end{aligned}$$

An improvement of this join algorithm may be achieved by overlapping the sorting of the two files. This can significantly reduce the execution time of the join if a "pipeline type" algorithm is employed. To illustrate this assumption, let us consider the simplified case where R and T both contain n pages and the number of processors is also n. In this case half of the time to sort R using the pipelined merge algorithm is for propagating the last page of R through the pipeline; now, as soon as the first processor has processed the nth page of R, it can start reading and processing pages of T. By using the pipeline again, we are able to replace $2T_{\text{sort}}(n)$ by $3/2T_{\text{sort}}(n)$ in the execution time for the join. Furthermore, since pages of T emerge one at a time from the pipeline, we may begin merging the 2 sorted relations as soon as the first page of the sorted T relation is produced.

Another possible improvement would be to use a parallel rather than a sequential scheme for joining the two

sorted files, if such a scheme could be designed. At this point, it is not clear to us if parallelism can improve the bound of $(n+m)$ I/O operations for the merge.

It should be noted that by using a merge sort algorithm to perform the join, we obtain a relation sorted with respect to the join attribute; this property might be desirable if the result relation is the final result of a query, or if it becomes the source relation for a subsequent join using the same joining attribute.

However, in the general case, the relative performance of a parallel sort-merge join and other parallel join methods are still unknown.

6.2.2. Control cost of parallel algorithms

Aside from the three major processing cost components that we have considered (computation, communication and I/O), several issues remain to be investigated for a more comprehensive evaluation of parallel algorithms. In particular, estimating the cost of controlling a complex parallel algorithm has been eluded by most researchers. Except in a few theoretical studies [Robi80, Vish81], no account is usually made for the cost of assigning (or reassigning) processors, or for the synchronization of processors at the initiation of a new stage of execution. While this cost may be negligible when parallel processors are

highly synchronized (in an SIMD architecture, for example), or when the parallel algorithm is simple, it might become a significant overhead when complex parallel sorting algorithms are executed. Controlling the execution of a parallel algorithm incurs both a communication overhead (for the exchange of messages between the controller and the processor) and processing overhead (for the controller to lookup and update various task and file tables). Thus, adequate tools are needed to accurately model this aspect in the analysis of parallel algorithms.

REFERENCES

- [Astr76]M.M. Astrahan, "System R: a Relational Approach to Database Management," ACM TODS 1, 2, (June 1976).
- [Babb79]E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS 4, 1, (March 1979).
- [Batc68]K.E. Batcher, "Sorting networks and their applications," Proceedings Spring Joint Computer Conference 32, pp. 307-314 (1968).
- [Baud78]G.M. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," IEEE Transactions on Computers c-27, 1, pp. 84-87 (January 1978).
- [Bent79]J.L. Bentley and H.T. Kung, "A tree machine for searching problems," Proceedings 1979 International Conference on Parallel Processing, pp. 257-266 (August 1979).
- [Blas77]M.W. Blasgen and K.P. Eswaran, "Storage and Access in Relational Data Bases," IBM System Journal 16, 4, (1977).
- [Bora80]H. Boral, D.J. DeWitt, D. Friedland, and W.K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," (Submitted October 1980).
- [Bora81]H. Boral and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," ACM TODS 6, (June 1981).
- [Corp80]IBM Corporation, "IBM 3380 Direct Access Storage description and User's Guide," IBM Document GA26-1664-0, File No. S/370-07,4300-07 (1980).
- [Daya82]U. Dayal, R.H. Katz, and N. Goodman, "An Extended Relational Algebra with Control over Duplicate Elimination," ACM Gigmod-Sigart Conference on Principles of Database Systems, (March 1982).
- [DeWi79]D.J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers c-28, 6, (June 1979).

- [DeWi81]D.J. DeWitt and P. Hawthorn, "A Performance Evaluation of Database Machine Architectures," Proceedings VLDB-7, (September 1981).
- [Even74]S. Even, "Parallelism in Tape Sorting," CACM 17, 4, (April 1974).
- [Fell68]W. Feller, An Introduction to Probability and its Applications. (1968).
- [Gavr75]F. Gavril, "Merging with Parallel Processors," CACM 18, 10, (October 1975).
- [Goke73]L.R. Goke and G.J. Lipovski, "Banyan networks for partitioning multiprocessor systems," 1st Annual Symposium on Computer Architecture, pp. 21-28 (December 1973).
-
- [Good81]J.R. Goodman, "An Investigation of Multiprocessor Structures and Algorithms for Database Management," Memo No. UCB/ERL M81/33, University of California, Berkeley (May 1981) Ph.D. Thesis.
- [Hirs78]D.S. Hirschberg, "Fast Parallel Sorting Algorithms," CACM 21, 8, (August 1978).
- [Kers80]L. Kerschberg, P.D. Ting, and S.B. Yao, "Query Optimization in Star Computer Networks," Bell Laboratories Database Research Report (March 1980).
- [Kim80]W. Kim, Query Optimization for Relational Database Systems, University of Illinois, Urbana-Champaign (1980) Ph. D. Thesis.
- [Leil78]H.O. Leilich, G. Stiege, and H. Ch. Zeidler, "A Search Processor for Data Base Management Systems," Proc 4th Conference on Very Large Databases, (1978).
- [Lin76]S.C. Lin, D.C.P. Smith, and J.M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications," ACM TODS 1, 1, (March 1976).
- [Lori71]H. Lorin, "A Guided Bibliography to Sorting," IBM Syst. J. 10, 3, (1971).
- [Mins72]N. Minsky, "Rotating Storage Devices as Partially Associative Memories," Proc FJCC, (1972).
- [Mull75]D.E. Muller and F.P. Preparata, "Bounds for Complexity of Networks for Sorting and for Switching," JACM, (April 1975).

- [Nass79]D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Transactions on Computers C-27, 1, (January 1979).
- [Ozka77.]E.A. Ozkarahan and K.C. Sevcik, "Analysis of Architectural Features for Enhancing the Performance of a Database Machine," ACM TODS 2, 4, (December 1977).
- [Prep78]F.P. Preparata, "New Parallel Sorting Schemes," IEEE Transactions on Computers c-27, 7, (July 1978).
- [Robi79]J.T. Robinson, "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms," IEEE Transactions on Software Engineering SE-5, 1, (January 1979).
- [Sieg77]H.J. Siegel, "The Universality of Various Types of SIMD Machine Interconnection Networks," Proceedings of the Fourth Annual Symposium on Parallel Architecture, (March 1977).
- [Slot70]D.L. Slotnick, "Logic Per Track Device," in Advances in Computers, ed. F. Alt, Academic Press, NY (1970).
- [Smit75]J.M. Smith and P. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," CACM 18, 10, (October 1975).
- [Spir76]P.M. Spira and I. Munro , "Sorting and Searching in Multisets," SIAM Journal Comp 5, 1, (March 1976).
- [Ston71.]H.S. Stone, "Parallel processing with the perfect shuffle," IEEE Transactions on Computers c-20, 2, pp. 153-161 (February 1971).
- [Ston78.]H.S. Stone, "Sorting on STAR," IEEE Transactions on Software Engineering 4, 2, (March 1978).
- [Ston76]M.R. Stonebraker, E. Wong, and P. Kreps, "The Design and Implementation of INGRES," ACM TODS 1, 3, (September 1976).
- [Ston81]M.R. Stonebraker, "Operating System Support for Database Management," CACM 24, 7, (July 1981).
- [Su79]S.Y.W. Su, "Cellular-Logic Devices: Concepts and Applications," IEEE Computer 12, 3, (March 1979).

- [Thom77]C.D. Thompson and H.T. Kung, "Sorting on a Mesh Connected Parallel Computer," CACM 20, 4, (April 1977).
- [Todd78]S. Todd, "Algorithm and Hardware for a Merge Sort Using Multiple Processors," IBM J. Res. Develop. 22, 5, (September 1978).
- [Vali75]L.G. Valiant, "Parallelism in Comparison Problems," SIAM J. of Computing 3, 4, (September 1975).
- [Vish81]U. Vishkin, Synchronized Parallel Computation, Technion Institute, Israel (1981) Ph.D. Thesis.
- [Voor71]D.C. Van Voorhis, , Stanford University (1971) Ph.D. Thesis.
-
- [Wirt76]N. Wirth, "Algorithms + Data Structures = Programs," in Prentice-Hall, (1976).
- [Yao79]B. Yao, "Optimization of Query Evaluation Algorithms," ACM TODS 4, 2, (June 1979).